VISIT…

Foundations of Combinatorics with Applications

Edward A. Bender
S. Gill Williamson

© 2005 E.A. Bender and S.G. Williamson All Rights Reserved

Contents

Contents iii

Preface ix

Part I Counting and Listing 1
Preliminary Reading 2

1 Basic Counting 5
Introduction 5
1.1 Lists with Repetitions Allowed 6
Using the Rules of Sum and Product 9
Exercises 10
1.2 Lists with Repetitions Forbidden 11
Exercises 17
1.3 Sets 19
*Error Correcting Codes 27
Exercises 30
1.4 Recursions 32
Exercises 35
1.5 Multisets 36
Exercises 38
Notes and References 39

2 Functions 41
Introduction 41
2.1 Some Basic Terminology 41
Terminology for Sets 41
What are Functions? 42
Exercises 44
2.2 Permutations 45
Exercises 50
2.3 Other Combinatorial Aspects of Functions 51
Monotonic Functions and Unordered Lists 51
Image and Coimage 54
The Pigeonhole Principle 55
Exercises 57
***2.4 Boolean Functions** 59
Exercises 63
Notes and References 64

3 Decision Trees 65

Introduction 65

3.1 Basic Concepts of Decision Trees 65
Exercises 73

*3.2 Ranking and Unranking 75
Calculating RANK 76
Calculating UNRANK 79
*Gray Codes 81
Exercises 83

*3.3 Backtracking 84
Exercises 90

Notes and References 91

4 Sieving Methods 93

Introduction 93
Structures Lacking Things 93
Structures with Symmetries 94

4.1 The Principle of Inclusion and Exclusion 94
Exercises 100
*Bonferroni's Inequalities 102
*Partially Ordered Sets 103
Exercises 104

4.2 Listing Structures with Symmetries 106
Exercises 109

4.3 Counting Structures with Symmetries 111
*Proofs 114
Exercises 116

Notes and References 117

Part II Graphs 119

5 Basic Concepts in Graph Theory 121

Introduction 121

5.1 What is a Graph? 121
Exercises 124

5.2 Equivalence Relations and Unlabeled Graphs 126
Exercises 130

5.3 Paths and Subgraphs 131
Exercises 134

5.4 Trees 136
Exercises 139

5.5 Directed Graphs (Digraphs) 141
Exercises 144

*5.6 Computer Representations of Graphs 146
Exercises 146

Notes and References 147

6 A Sampler of Graph Topics 149

Introduction 149

6.1 Spanning Trees 150
Minimum Weight Spanning Trees 150
Lineal Spanning Trees 153
Exercises 155

6.2 Coloring Graphs 157
Exercises 161

6.3 Planar Graphs 162
Euler's Relation 163
Exercises 164
The Five Color Theorem 165
Exercises 166
*Algorithmic Questions 167
Exercises 169

6.4 Flows in Networks 170
The Concepts 170
An Algorithm for Constructing a Maximum Flow 173
Exercises 176
*Cut Partitions and Cut Sets 176
Exercises 179

*6.5 Probability and Simple Graphs 180
Exercises 186

6.6 Finite State Machines 188
Turing Machines 188
Finite State Machines and Digraphs 189
Exercises 192

Notes and References 194

Part III Recursion 195

7 Induction and Recursion 197

Introduction 197

7.1 Inductive Proofs and Recursive Equations 198
Exercises 203

7.2 Thinking Recursively 204
Exercises 209

7.3 Recursive Algorithms 210
Obtaining Information: Merge Sorting 210
Local Descriptions 212
*Computer Implementation 215
Exercises 217

7.4 Divide and Conquer 220
Exercises 223

Notes and References 225

8 Sorting Theory 227

Introduction 227

8.1 Limits on Speed 228
Motivation and Proof of the Theorem 229
Exercises 231

8.2 Software Sorts 232
Binary Insertion Sort 233
Bucket Sort 234
Merge Sorts 235
Quicksort 235
Heapsort 236
Exercises 237

8.3 Sorting Networks 238
8.3.1 Speed and Cost 238
Parallelism 239
How Fast Can a Network Be? 240
How Cheap Can a Network Be? 240
Exercises 240
8.3.2 Proving That a Network Sorts 241
The Batcher Sort 243
Exercises 244

Notes and References 245

9 Rooted Plane Trees 247

Introduction 247

9.1 Traversing Trees 248
Depth First Traversals 249
Exercises 251

9.2 Grammars and RP-Trees 253
Exercises 258

*__9.3 Unlabeled Full Binary RP-Trees__ 259
Exercises 265

Notes and References 266

Part IV Generating Functions 267

10 Ordinary Generating Functions 269

Introduction 269

10.1 What are Generating Functions? 269
Exercises 273

10.2 Solving a Single Recursion 275
Exercises 280

10.3 Manipulating Generating Functions 282
Obtaining Recursions 282
Derivatives, Averages and Probability 284
Exercises 291

10.4 The Rules of Sum and Product 291
Exercises 298

Notes and References 304

*11 Generating Function Topics 307

Introduction 307

11.1 Systems of Recursions 308
Exercises 313

11.2 Exponential Generating Functions 315
The Exponential Formula 320
Exercises 326

11.3 Symmetries and Pólya's Theorem 330
Exercises 338

11.4 Asymptotic Estimates 339
Recursions 341
Sums of Positive Terms 344
Generating Functions 349
Exercises 355

Notes and References 359

Appendix A Induction 361
Exercises 365

Appendix B Rates of Growth and Analysis of Algorithms 367

B.1 The Basic Functions 368
Exercises 374

B.2 Doing Arithmetic 376

B.3 NP-Complete Problems 377
Exercises 379

Notes and References 380

Appendix C Basic Probability 381

C.1 Probability Spaces and Random Variables 381

C.2 Expectation and Variance 384

Appendix D Partial Fractions 387
Theory 387
Computations 388

Solutions to Odd Exercises and Most Appendix Exercises 393

Index 461

Preface

Combinatorics, the mathematics of the discrete, has blossomed in this generation. On the theoretical side, a variety of tools, concepts and insights have been developed that allow us to solve previously intractable problems, formulate new problems and connect previously unrelated topics. On the applied side, scientists from physicists to biologists have found combinatorics essential in their research. In all of this, the interaction between computer science and mathematics stands out as a major impetus for theoretical developments and for applications of combinatorics. This text provides an introduction to the mathematical foundations of this interaction and to some of its results.

Advice to Students

This book does not assume any previous knowledge of combinatorics or discrete mathematics. Except for a few items which can easily be skipped over and some of the material on "generating functions" in Part IV, calculus is not required. What is required is a certain level of ability or "sophistication" in dealing with mathematical concepts. The level of mathematical sophistication that is needed is about the same as that required in a solid beginning calculus course.

You may have noticed similarities and differences in how you think about various fields of mathematics such as algebra and geometry. In fact, you may have found some areas more interesting or more difficult than others partially because of the different thought patterns required. The field of combinatorics will also require you to develop some new thought patterns. This can sometimes be a difficult and frustrating process. Here is where patience, mathematical sophistication and a willingness to ask "stupid questions" can all be helpful.

Combinatorics differs as much from mathematics you are likely to have studied previously as algebra differs from geometry. Some people find this disorienting and others find it fascinating. The introductions to the parts and to the chapters can help you orient yourself as you learn about combinatorics. Don't skip them.

Because of the newness of much of combinatorics, a significant portion of the material in this text was only discovered in this generation. Some of the material is closely related to current research. In contrast, the other mathematics courses you have had so far probably contained little if anything that was not known in the Nineteenth Century. Welcome to the frontiers!

The Material in this Book

Combinatorics is too big a subject to be done justice in a single text. The selection of material in this text is based on the need to provide a solid introductory course for our students in pure mathematics and in mathematical computer science. Naturally, the material is also heavily influenced by our own interests and prejudices.

Parts I and II deal with two fundamental aspects of combinatorics: enumeration and graph theory. "Enumeration" can mean either counting or listing things. Mathematicians have generally limited their attention to counting, but listing plays an important role in computer science, so we discuss both aspects. After introducing the basic concepts of "graph theory" in Part II, we present

a variety of applications of interest in computer science and mathematics. Induction and recursion play a fundamental role in mathematics. The usefulness of recursion in computer science and in its interaction with combinatorics is the subject of Part III. In Part IV we look at "generating functions," a powerful tool for studying counting problems. We have included a variety of material not usually found in introductory texts:

- Trees play an important role. Chapter 3 discusses decision trees with emphasis on ranking and unranking. Chapter 9 is devoted to the theory and application of rooted plane trees. Trees have many practical applications, have an interesting and accessible theory and provide solid examples of inductive proofs and recursive algorithms.
- Software and network sorts are discussed in Chapter 8. We have attempted to provide the overview and theory that is often lacking elsewhere.
- Part IV is devoted to the important topic of generating functions. We could not, in good conscience, deny our students access to the more combinatorial approaches to generating functions that have emerged in recent years. This necessitated a longer treatment than a quick ad hoc treatment would require. Asymptotic analysis of generating functions presented a dilemma. On the one hand, it is very useful; while on the other hand, it cannot be done justice without an introductory course in complex analysis. We chose a somewhat uneasy course: In the last section we presented some rules for analysis that usually work and can be understood without a knowledge of complex variables.

Planning a Course

A variety of courses can be based on this text. Depending on the material covered, the pace at which it is done and the level of rigor required of the students, this book could be used in a challenging lower division course, in an upper division course for engineering, science or mathematics students, or in a beginning graduate course. There are a number of possibilities for choosing material suitable for each of these classes. A graduate course could cover the entire text at a leisurely pace in a year or at a very fast pace in a semester. Here are some possibilities for courses with a length of one semester to two quarters, depending on how much parenthesized optional material is included. Parts of an optional chapter can also be used instead of the entire chapter.

- A lower division course: 1, 2.1–2.3, (2.4), 3.1, (4.1), 5.1, (5.2), 5.3–5.5, (6), 7.1, 7.2, (7.3), (8), 9.1, (9.2).
- An upper division or beginning graduate course emphasizing mathematics: 1–3, 4.1, (4.2), 4.3, 5, 6.1, (6.2–6.4), 7, (8) 9.1, (9.2–9.3), 10, (11).
- An upper division or beginning graduate course emphasizing computer science: 1–3, 4.1, 5, 6.1, 6.3, (6.4), (6.5), 7, 8, (9.1), 9.2, 9.3, 10, (11.4).

Asterisks, or stars, (*) appear before various parts of the text to help in course design. Starred exercises are either more difficult than other exercises in that section or depend on starred material. Starred examples are generally more difficult than other material in the chapter. A section or chapter that is not as central as the rest of the material is also starred. The material in Part IV, especially parts of Chapter 11, is more difficult than the rest of the text.

Special thanks are due Fred Kochman whose many helpful comments have enhanced the readability of this manuscript and reduced its errors. This manuscript was developed using TEX, Donald E. Knuth's impressive gift to technical writing.

PART I

Counting and Listing

Enumerative combinatorics deals with listing and counting the elements in finite sets. Why is this of interest? Determining the number of elements in a finite set is the fundamental tool in the analysis of the running times and space demands of computer algorithms. It is also of importance in various areas of science, most notably statistical mechanics and structural chemistry, and in some areas of mathematics. Listing elements plays a role in the design of algorithms and in structural chemistry as well as other areas. In addition, the questions may be interesting in themselves; that is, some people find questions of the form "How many structures are there such that ... ?" interesting.

In this part we'll study some fundamental counting and listing techniques. These tools are useful throughout combinatorics and many of them are essential for other topics that are covered later in the text. In particular, the Rules of Sum and Product form the basis of practically all of our counting calculations. The set theoretic notation and terminology that we introduce in the first two chapters is also important for the remainder of the text.

Here are some examples of the types of problems that our tools will enable us to solve systematically.

Example 1 Birthdays There are 30 students in a classroom. What is the probability that all of them have different birthdays? Let's assume that people are equally likely to be born on each day of the year and that a year has 365 days. (Neither assumption is quite correct.) Suppose we could determine the number of possible ways birthdays could be assigned to 30 people, say N, and the number of ways this could be done so that all the birthdays are different, say D. Our answer would be D/N. □

Example 2 Names In a certain land on a planet in a galaxy far away the alphabet contains only 5 letters which we will transliterate as A, I, L, S and T in that order. All names are 6 letters long, begin and end with consonants and contain two vowels which are not adjacent to each other. Adjacent consonants must be different. How many possible names are there? Devise a systematic method for listing them in dictionary order. The list begins with LALALS, LALALT, LALASL, LALAST, LALATL, LALATS, LALILS and ends with TSITAT, TSITIL, TSITIS, TSITIT. □

Example 3 Data storage An ecologist plans to simulate on a computer the growth of a biological community containing 6 competing species. He plans to try numerous variations in the environment. After each simulation, he'll order the species from most abundant to least abundant. He wants to keep track of how often each ordering occurs and, after his simulations are over, manipulate the collection of counts in various ways. How can he index his storage in a compact manner? □

Example 4 Arrangements We have 32 identical dominoes with no marks on them and a chessboard. Each domino will exactly cover two squares of the chessboard. How many ways can we arrange the dominoes so that they cover the entire board? □

Example 5 Symmetries We have a cube, some red paint and some green paint. How many different ways can we paint the cube so that each face is either all red or all green? Obtain a list of all the different ways. A first approach to this problem might be to consider coloring the first face red or green, then the second red or green and so on until we have a list of all the possibilities. This ignores an important fact about the problem: a cube looks the same from many points of view; i.e., it has symmetries. For example, there is only one way to color a cube so that it has one red face and five green ones. □

In the next four chapters, we'll study some fundamental techniques for counting and listing structures. What do we mean by "structures?" It is simply a general term for whatever things we are counting. We chose it rather than "thing" or "object" to emphasize that what we are counting has some internal organization. If the things we were trying to list or count did not have some sort of internal organization, we would have no way to systematically analyze them. A list of 30 distinct birthdays or a cube with colored faces are two examples of structures. Such a list has an internal organization: For the birthdays, we have 30 distinct days of the year written in some order. A cube has considerable structure because it can be rotated in various ways and end up occupying the same space.

Understanding the internal organization clearly is the first step in solving a counting or listing problem. For the birthdays, the organization is a sequence of 30 distinct numbers between 1 and 365 inclusive. For the cube, the organization is somehow tied to the cube's symmetries. We'll easily see how to answer the birthday question thanks to the simple description of the structure's organization. We have problems with the cube because we haven't yet come up with a clear description.

In Chapter 1 we'll study some simple structures—ordered and unordered lists with and without repetitions—and we'll introduce tools for counting them. The main tools are the Rules of Sum and Product. Recursions and generating functions will also appear briefly. We'll return to generating functions in Part IV.

In Chapter 2 we'll study functions and "permutations." Besides being of interest in themselves, functions provide another way to look at the material in Chapter 1, and permutations are essential for Chapter 4. We conclude Chapter 2 with a discussion of Boolean functions and combinatorial logic.

It is frequently necessary to generate combinatorial objects or accumulate information about them rather than just count them. In Chapter 3 you'll see how to use the function viewpoint and "trees" to generate lists of structures. Furthermore, we'll study how to access particular items in the list using "ranking." This is what we need for the biologist's problem. Trees are an important structure in computer science, so you'll encounter them again in this book.

In Chapter 4 we'll study two rather unrelated topics that did not merit a separate chapter. The first topic is counting and listing structures with symmetries. Our earlier notion of permutation provides the foundation for this discussion. The second topic is a method of counting structures in a somewhat indirect manner, called the "Principle of Inclusion and Exclusion."

Preliminary Reading

At various points in our discussion we will need to make use of proof by induction. In fact, induction is a more common proof technique in combinatorics than in most other branches of mathematics. We recommend that you review proof by induction in Appendix A (p. 361).

At times we will estimate values using "Big-Oh" and "little-oh" notation as well as the notation $f(n) \sim g(n)$. These are discussed in Section B.1 (p. 368). You may wish to look at this section quickly now and refer back to it as needed.

Since probability is a natural adjunct to counting, we'll encounter it from time to time in the examples and homework. The necessary background is reviewed in Appendix C (p. 381). You should look this over now and refer back to it as needed.

The algebraic rules for operating with sets are also familiar to most beginning university students. Here is such a list of the basic rules. In each case the standard name of the rule is given first, followed by the rule as applied first to $\cap$ and then to $\cup$.

Theorem 0.1 Algebraic rules for sets *The universal set U is not mentioned explicitly but is implicit when we use the notation $\sim X = U - X$ for the complement of X. An alternative notation is $X^c = \sim X$.*

Associative:	$(P \cap Q) \cap R = P \cap (Q \cap R)$	$(P \cup Q) \cup R = P \cup (Q \cup R)$
Distributive:	$P \cap (Q \cup R) = (P \cap Q) \cup (P \cap R)$	$P \cup (Q \cap R) = (P \cup Q) \cap (P \cup R)$
Idempotent:	$P \cap P = P$	$P \cup P = P$
Double Negation:	$\sim\sim P = P$	
DeMorgan:	$\sim(P \cap Q) = \sim P \cup \sim Q$	$\sim(P \cup Q) = \sim P \cap \sim Q$
Absorption:	$P \cup (P \cap Q) = P$	$P \cap (P \cup Q) = P$
Commutative:	$P \cap Q = Q \cap P$	$P \cup Q = Q \cup P$

CHAPTER 1

Basic Counting

Introduction

Before beginning, we must confront some matters of notation. Two words that we shall often use are *set* and *list*. Both words refer to collections of objects. There is no standard notation for lists. Some of those in use are

	apple banana pear peach	*a list of four items ...*
	apple, banana, pear, peach	*commas added for clarity ...*
and	(apple, banana, pear, peach)	*parentheses added.*

The notation for sets is standard: the items are separated by commas and surround by curly brackets as in

$$\{\text{apple}, \text{banana}, \text{pear}, \text{peach}\}.$$

The curly bracket notation for sets is so well established that you can normally assume it means a set—but beware, Mathematica® uses curly brackets for lists.

What is the difference between a set and a list? Quite a bit, and nothing. "Set" means a collection of distinct objects in which the order doesn't matter. Thus

$$\{\text{apple}, \text{peach}, \text{pear}\} \quad \text{and} \quad \{\text{peach}, \text{apple}, \text{pear}\}$$

are the same sets, and the set $\{\text{apple}, \text{peach}, \text{apple}\}$ is the same as the set $\{\text{apple}, \text{peach}\}$. In other words, repeated elements are treated as if they occurred only once. Thus two sets are the same if and only if each element that is in one set is in both. In a list, order is important and repeated objects are usually allowed. Thus

$$(\text{apple}, \text{peach}) \quad (\text{peach}, \text{apple}) \quad \text{and} \quad (\text{apple}, \text{peach}, \text{apple})$$

are three different lists. Two lists are the same if and only if they have exactly the same items in exactly the same positions. Thus, sets and lists are different.

On the other hand, people talk about things like "unordered lists," "sets with repetition," and so on. In fact, a set with repetition is so common that it has a name: *multiset*. Two multisets are the same if and only if each item that occurs exactly k times in one of them occurs exactly k times in both. In summary

- *list*: an ordered sequence (repeats allowed),
- *set*: a collection of distinct objects where order does not matter,
- *multiset*: a collection of objects (repeats allowed) where order does not matter.

Thus, an ordered set with repetition allowed is a list and an unordered list of distinct elements is a set. Whenever we refer to a list, we will indicate whether the elements must be distinct. Unless we

say otherwise, a list is ordered. An ordered list is sometimes called a *string*, a *sequence* or a *word*. A list is also called a *sample* or a *selection*, especially in probability and statistics. Lists are sometimes called vectors and the elements components.

The terminology "k-list" is frequently used in place of the more cumbersome "k long list." Similarly, we use k-set and k-multiset. Vertical bars (also used for absolute value) are used to denote the number of elements in a set or in a list. For example, if S is an n-set, then $|S| = n$.

We want to know how many ways we can do various things with a set. Here are some examples, which we illustrate by using the set $S = \{x, y, z\}$.

1. How many ways can we *list*, without repetition, all the elements of S? This means, how many ways can we arrange the elements of S in an (ordered) list so that each element of S appears exactly once in each of the lists. For the illustration, there are six ways: xyz, xzy, yxz, yzx, zxy and zyx. (These are all called permutations of S. People often use Greek letters like π and σ to indicate a permutation of a set.)
2. How many ways can we construct a k-list of distinct elements from the set? When $k = |S|$, this is the previous question. If $k = 2$ in the illustration, there are six ways: xy, xz, yx, yz, zx and zy.
3. If the list in the previous question is allowed to contain repetitions, what is the answer? There are nine ways for the illustration: xx, xy, xz, yx, yy, yz, zx, zy and zz.
4. If, in Questions 2 and 3, the order in which the elements appear in the list doesn't matter, what are the answers? For the illustration, the answers are three and six, respectively.
5. How many ways can the set S be partitioned into a collection of k pairwise disjoint nonempty smaller sets? With $k = 2$, the illustration has three such: $\{\{x\}, \{y, z\}\}$, $\{\{x, y\}, \{z\}\}$ and $\{\{x, z\}, \{y\}\}$.

We'll learn how to answer these questions without going through the time-consuming process of constructing (listing) all the items in question as we did for our illustration. Our answer to the last question will be somewhat unsatisfactory. Other answers to it will be discussed in later chapters.

1.1 Lists with Repetitions Allowed

How many ways can we construct a k-list (repeats allowed) using an n-set? Look at our illustration in Question 3 above. The first entry in the list could be x, y or z. After any of these there were three choices (x, y or z) for the second entry. Thus there are $3 \times 3 = 9$ ways to construct such a list. The general pattern should be clear: There are n ways to choose each list entry. Thus

Theorem 1.1 *There are n^k ways to construct a k-list from an n-set.*

This calculation illustrates an important principle:

Theorem 1.2 Rule of Product *Suppose structures are to be constructed by making a sequence of k choices such that, (i) the ith choice can be made in c_i ways, a number independent of what choices were made previously, and (ii) each structure arises in exactly one way in this process. Then, the number of structures is $c_1 \times \cdots \times c_k$.*

"Structures" as used above can be thought of simply as elements of a set. We prefer the term structures because it emphasizes that the elements are built up in some way; in this case, by making a sequence of choices. In the previous calculation, the structures are lists of k things which are built up by adding one thing at a time. Each thing is chosen from a given set of n things and $c_1 = c_2 = \ldots = c_k = n$.

Definition 1.1 Cartesian Product *If $C_1, \ldots, C_k$ are sets, the* **Cartesian product** *of the sets is written $C_1 \times \cdots \times C_k$ and consists of all k-lists $(x_1, \ldots, x_k)$ with $x_i \in C_i$ for $1 \leq i \leq k$.*

A special case of the Rule of Product is the fact that the number of elements in $C_1 \times \cdots \times C_k$ is the product $|C_1| \cdots |C_k|$. Here C_i is the collection of ith choices and $c_i = |C_i|$. This is only a special case because the Rule of Product would allow the *collection* C_i to depend on the previous choices $x_1, \ldots, x_{i-1}$ as long as the *number* c_i of possible choices does not depend on $x_1, \ldots, x_{i-1}$. The last example in Appendix A gives a proof of this special case of the Rule of Product. In fact, that proof can be altered to give a proof of the general case of the Rule of Product. We will not do so.

Here is a property associated with Cartesian products that we will find useful in our later discussions.

Definition 1.2 Lexicographic Order *If $C_1, \ldots, C_k$ are ordered lists of distinct elements, we may think of them as sets and form the Cartesian product $P = C_1 \times \cdots \times C_k$. The* **lexicographic order** *on P is defined by saying that $a_1 \ldots a_k < b_1 \ldots b_k$ if and only if there is some $t \leq k$ such that $a_i = b_i$ for $i < t$ and $a_t < b_t$.*

Often we say *lex order* instead of lexicographic order. If all the C_i's equal $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$, then lex order is simply numerical order of k digit integers with leading zeroes allowed. Suppose that all the C_i's equal (<space>, A, B, ..., Z). If we throw out those elements of P that have a letter following a space, the result is dictionary order. Unlike these two simple examples, the C_i's usually vary with i.

Example 1.1 A simple count The North-South streets in Rectangle City are named using the numbers 1 through 12 and the East-West streets are named using the letters A through H. Thus, the most southwesterly intersection occurs where First and A streets meet. How many blocks are within the city?

We may think of the city of as consisting of rows of blocks. Each row contains the blocks encountered as we cross the city from East to West. The number of rows is the number of rows of blocks encountered as we cross the city from North to South. This is much like the rows and columns of a matrix. We can apply the Rule of Product: Choose a row and then choose a block in that row. What answer does this give? If you think it is $12 \times 8 = 96$, you're almost correct. Read on.

Each block can be labeled by the streets at its southwesterly corner. These labels have the form (x, y) where x is between 1 and 11 inclusive and y is between A and G. (If you don't see why 12 and H are missing, draw a picture and look at southwesterly corners.) By the Rule of Product there are $11 \times 7 = 77$ blocks. In this case the structures can be taken to be the descriptions of the blocks. Each description has two parts: the names of the north-south and East-West streets at the block's southwest corner. □

Example 1.2 Counting names We now return to the faraway galaxy that was mentioned in Example 2 (p. 1).

The possible positions for the two vowels are $(2, 4)$, $(2, 5)$ and $(3, 5)$. Each of these results in two isolated consonants and two adjacent consonants. Thus the answer is the product of the following factors:

- choose the vowel locations (3 ways);
- choose the vowels (2×2 ways);
- choose the isolated consonants (3×3 ways);
- choose the adjacent consonants (3×2 ways).

The answer is 648. This construction can be interpreted as a Cartesian product as follows. C_1 is the set of lists of possible positions for the vowels, C_2 is the set of lists of vowels in those positions, and C_3 and C_4 are sets of lists of consonants. Thus

$$C_1 = \{(2,4),(2,5),(3,5)\} \qquad C_2 = \{\text{AA}, \text{AI}, \text{IA}, \text{II}\}$$
$$C_3 = \{\text{LL}, \text{LS}, \text{LT}, \text{SL}, \text{SS}, \text{ST}, \text{TL}, \text{TS}, \text{TT}\} \qquad C_4 = \{\text{LS}, \text{LT}, \text{SL}, \text{ST}, \text{TL}, \text{TS}\}.$$

For example, ((2,5), IA, SS, ST) in the Cartesian product corresponds to the word SISTAS. □

Here's another important principle, the proof of which is self evident:

Theorem 1.3 Rule of Sum *Suppose a set T of structures can be partitioned into sets $T_1, \ldots, T_j$ so that each structure in T appears in exactly one T_i, then*

$$|T| = |T_1| + \cdots + |T_j|.$$

Example 1.3 Counting names (revisited) We'll redo the previous example using this principle.

The possible vowel (V) and consonant (C) patterns for names are CCVCVC, CVCCVC and CVCVCC. Since these patterns are disjoint and cover all cases, we must compute the number of names of each type and add the results together. For the first pattern we have a product of six factors, one for each choice of a letter: $3 \times 2 \times 2 \times 3 \times 2 \times 3 = 216$. The other two patterns also give 216, for a total of 648 names.

This approach has a wider range of applicability than the method we used in the previous example. We were only able to avoid the Rule of Sum in the first method because each pattern contained the same number of vowels, isolated consonants and adjacent consonants. Here's an example that requires the Rule of Sum. Suppose a name consists of only four letters, namely two vowels and two consonants, constructed so that the vowels are not adjacent and, if the consonants are adjacent, then they are different. There are four patterns: CVCV, VCVC, VCCV. By the Rule of Product, the first two are each associated with 36 names, but VCCV is associated with only 24 names because of the adjacent consonants. Hence, we cannot choose a pattern and then proceed to choose vowels and consonants. On the other hand, we can apply the Rule of Sum to get a total of 96 names. □

Example 1.4 Smorgasbord College committees Smorgasbord College has four departments which have 6, 35, 12 and 7 faculty members. The president wishes to form a faculty judicial committee to hear cases of student misbehavior. To avoid the possibility of ties, the committee will have three members. To avoid favoritism the committee members will be from different departments and the committee will change daily. If the committee only sits during the normal academic year (165 days), how many years can pass before a committee must be repeated?

If T is the set of all possible committees, the answer is $|T|/165$. Let T_i be the set of committees with no members from the ith department. By the Rule of Sum $|T| = |T_1| + |T_2| + |T_3| + |T_4|$. By the Rule of Product

$$|T_1| = 35 \times 12 \times 7 = 2940 \qquad |T_3| = 35 \times 6 \times 7 = 1470$$
$$|T_2| = 6 \times 12 \times 7 = 504 \qquad |T_4| = 35 \times 12 \times 6 = 2520.$$

Thus the number of years is $7434/165 = 45^{+}$. Due to faculty turnover, a committee need never repeat—if the president's policy lasts that long. □

Using the Rules of Sum and Product

Whenever we encounter a new technique, there are two questions that arise:

- *When* is it used?
- *How* is it used?

For the Rules of Sum and Product, the answers are intertwined:

Technique Rules for AND and OR *Suppose you wish to count the number of structures in a set and that you can describe how to construct the structures in terms of subconstructions that are connected by "ands" and "ors." If this leads to the construction of each structure in a unique way, then the Rules of Sum and Product apply. To use them, replace "ands" by products and "ors" by sums. Whenever you write something like "Do A AND do B," it should mean "Do A AND THEN do B" because the Rule of Product requires that the choices be made sequentially. We will usually omit "then".*

Example 1.5 Applying the technique To see how this technique is applied, let's look back at Example 1.4. A committee consists of either

- One person from Dept. 1 AND one person from Dept. 2 AND one person from Dept. 3, OR
- One person from Dept. 1 AND one person from Dept. 2 AND one person from Dept. 4, OR
- One person from Dept. 1 AND one person from Dept. 3 AND one person from Dept. 4, OR
- One person from Dept. 2 AND one person from Dept. 3 AND one person from Dept. 4.

The number of ways to choose a person from a department equals the number of people in the department. □

Until you become comfortable using the Rules of Sum and Product, look for "and" and "or" in what you do. This is an example of the *divide and conquer* tactic: break the problem into parts and work on each piece separately. Here the first part is getting a phrasing with "ands" and "ors;" the second part is calculating each of the individual pieces; and the third part is applying the Rules of Sum and Product.

Example 1.6 Palindromes A *palindrome* is a list that reads the same from right to left as it does from left to right. For example, ignoring capitalization, punctuation and spaces, "Madam I'm Adam." becomes the palindrome madamimadam.

How many k-long palindromes can be formed from an n-set? The first $\lceil k/2 \rceil$ list elements are arbitrary and the remaining elements are determined.* Thus the answer is $n^{\lceil k/2 \rceil}$.

Imagine a necklace of beads with a clasp. How many k-bead necklaces can be formed if we are given n different colors of round beads. When the necklace is worn we can tell the end of the necklace because of the clasp, but we can't distinguish a left end versus a right end. We can think of this as k-long lists where we consider two lists the same if one can be obtained from the other by reversing the list. If a list is a palindrome, it contributes one to the count. If a list is not a palindrome, the list and its reversal together contribute one to the count.

Let p be the number of palindrome lists and q the number of non-palindrome lists. We want $p + q/2$. The number of lists is $p + q$, which equals n^k and the number of palindromes is $n^{\lceil k/2 \rceil}$. Thus

$$p + q \;=\; n^k \qquad \text{and} \qquad p \;=\; n^{\lceil k/2 \rceil}$$

* The notation $\lceil x \rceil$ means least integer not less than x (that is, round up). For example $\lceil \pi \rceil = 4$.

and so $q = n^k - n^{\lceil k/2 \rceil}$. Finally we obtain our answer:

$$p + q/2 \;=\; n^{\lceil k/2 \rceil} + \frac{n^k - n^{\lceil k/2 \rceil}}{2} \;=\; \frac{n^{\lceil k/2 \rceil} + n^k}{2}. \;\square$$

Example 1.7 Listing instead of counting Suppose we want to write a program to actually list the things in a set T rather than just counting them. Instead of computing $|T|$, we have to execute a program that lists all items $t \in T$. What about the Rules of Sum and Product? The Rule of Sum becomes

```
For each t_1 ∈ T_1: list t_1.
For each t_2 ∈ T_2: list t_2.
...
For each t_j ∈ T_j: list t_j.
```

The Rule of Product becomes

```
For each first choice d_1:
   ...
      For each kth choice d_k:
         List the structure arising from the choices d_1, ..., d_k.
      End for
   ...
End for
```

This is actually more general than Theorem 1.2 since, in the code, the number of choices in each loop may depend on previous choices. See Chapter 3 for more discussion. $\square$

Exercises

In each of the exercises, indicate how you are using the Rules of Sum and Product. You can do this with the AND/OR technique.

1.1.1. How many different three digit positive integers are there? (No leading zeroes are allowed.) How many positive integers with at most three digits? What are the answers when "three" is replaced by "n?"

1.1.2. A small neighboring country of the one we revisited in Example 1.3 has the same alphabet and the same rules of formation, but names are only five letters long. How many names are possible?

1.1.3. Prove that the number of subsets of a set S, including the empty set and S itself, is $2^{|S|}$.
Hint. For each element of S you must make one of two choices: "x is/isn't in the subset."

1.1.4. A *composition* of a positive integer n is an ordered list of positive integers (called *parts*) that sum to n. The four compositions of 3 are 3; 2,1; 1,2 and 1,1,1.

(a) By considering ways to insert plus signs and commas in a list of n ones, obtain a formula for the number of compositions of n.
Hint. The four compositions above correspond to 1+1+1; 1+1,1; 1,1+1 and 1,1,1, respectively.

(b) Prove that the average number of parts in a composition of n is $(n+1)/2$.
Hint. Reverse the roles of "+" and "," and then look at the number of parts in the original and role-reversed compositions.

*1.1.5. In Example 1.3 we found that there were 648 possible names. Suppose that these are listed in the usual dictionary order. What is the last word in the first half of the dictionary (the 324th word)? the first word in the second half?

1.2 Lists with Repetitions Forbidden

What happens if we do not allow repeats in our list? Suppose we have n elements to choose from and wish to form a k-list with no repeats. How many lists are there?

We can choose the first entry in the list AND choose the second entry AND $\cdots$ AND choose the kth entry. There are $n-i+1$ ways to choose the ith entry since $i-1$ elements have been removed from the set to make the first part of the list. By the Rule of Product, the number of lists is $n(n-1)\cdots(n-k+1)$. Using the notation $n!$ for the product of the first n integers and writing $0!=1$, you should be able to see that this answer can be written as $n!/(n-k)!$, which is often designated by $(n)_k$ and called the *falling factorial*. We have proven

Theorem 1.4 *When repeats are not allowed, there are $n!/(n-k)! = (n)_k$ k-lists that can be constructed from an n-set.*

When $k=n$, a list without repeats is simply a *linear ordering* of the set. We frequently say "ordering" instead of "linear ordering." An ordering is sometimes called a "permutation" of S. Thus, we have proven that a set S can be (linearly) ordered in $|S|!$ ways.

Example 1.8 Lists without repeats How many lists without repeats can be formed from a 5-set? There are $5!=120$ 5-lists without repeats, $5!/1!=120$ 4-lists without repeats, $5!/2!=60$ 3-lists, $5!/3!=20$ 2-lists and $5!/4!=5$ 1-lists. By the Rule of Sum, this gives a total of 325 lists, or 326 if we count the empty list. In Exercise 1.2.11 you are asked to obtain an estimate when "5-set" is replaced with "n-set".

Suppose we have a problem involving k-lists with repeats allowed and we want the formula when repeats are not allowed. Since allowing repeats leads to powers and forbidding repeats leads to falling factorials, we might try to replace powers with falling factorials. Doing this without thinking, can easily give the wrong answers. Look back at Example 1.6 where we needed to count palindromes and obtained the formula $p=n^{\lceil k/2\rceil}$. Except for 1-long lists, a palindrome has repeated elements; for example, the first and last elements are equal. Thus we obtain $p=n$ when $k=1$ and $p=0$ when $k>1$ for palindromes without repeats. □

Lists can appear in many guises. In this next example, the people could be thought of as the positions in a list and the seats the things in the list. Sometimes it helps to find a reinterpretation like this for a problem. At other times it is easier to tackle the problem starting over again from scratch. These methods can lead to several approaches to a problem. That can make the difference between a solution and no solution or between a simple solution and a complicated one. You should practice using both methods, even on the same problem.

Example 1.9 Linear arrangements How many different ways can 100 people be arranged in the seats in a classroom that has exactly 100 seats?

Each seating is simply an ordering of the people. Thus the answer is 100!. Simply writing 100! probably gives you little idea of the size of the number of seatings. A useful approximation for factorials is given by Stirling's formula:

Theorem 1.5 Stirling's formula *$\sqrt{2\pi n}\,(n/e)^n$ approximates $n!$ with a relative error under $1/10n$.*

We say that $f(x)$ approximates $g(x)$ with a *relative error* at most $\delta(x)$ if $|f(x)/g(x) - 1| \le \delta(x)$. Thus, the theorem states that $\sqrt{2\pi n}\,(n/e)^n/n!$ differs from 1 by less than $1/10n$. When relative error is multiplied by 100, we obtain "percentage error." If we simply want to note that the relative error goes to 0 as $n \to \infty$, we can write[1]

$$n! \;\sim\; \sqrt{2\pi n}(n/e)^n \quad \text{or, equivalently,} \quad n! \;=\; \sqrt{2\pi n}(n/e)^n(1 + o(1)).$$

This is weaker than Theorem 1.5 because $o(1)$ stands for something that can be replaced by *some* function $h(n)$ with $\lim_{n\to\infty} h(n) = 0$, but the theorem tells us more, namely the function $h(n)$ is so small that $|h(n)| < 1/10n$.

By Stirling's formula, we find that 100! is nearly 9.32×10^{157}, which is much larger than estimates of the number of atoms in the universe.

Now suppose we still have 100 seats but have only 95 people. We need to think a bit more carefully than before. One approach is to put the people in some order (e.g., alphabetical), select a list of 95 seats, and then pair up people and seats so that the first person gets the first seat, the second person the second seat, and so on. By the general formula for lists without repetition, the answer is $100!/(100-95)! = 100!/120$. We can also solve this problem by thinking of the people as positions in a list and the seats as entries. Do it. □

The next example is starred because it is above the level of this chapter; therefore you may want to just skim it or maybe even omit it. It illustrates some of the calculations that one often runs into in obtaining estimates for large values of n and obtains the useful formula (1.2).

***Example 1.10 Estimating $n!/(n-k)!$** This example requires familiarity with the notations $O(\,)$ and $o(\,)$, which are discussed in Appendix B.

Suppose we want to estimate the number of k-lists without repeats that can be formed from an n-set; that is, we want to estimate $n!/(n-k)!$. In this example, we're interested in obtaining the estimate when n is large and k is much smaller than n. Of course, we can use Stirling's formula, which gives us the estimate

$$\frac{n!}{(n-k)!} \;\sim\; \frac{\sqrt{2\pi n}(n/e)^n}{\sqrt{2\pi(n-k)}((n-k)/e)^{n-k}} \;=\; \frac{n^{n+1/2}e^{-k}}{(n-k)^{n-k+1/2}}.$$

This is still rather messy. How can we simplify it? We have

$$\frac{n^{n+1/2}}{(n-k)^{n-k+1/2}} \;=\; n^k\left(\frac{n}{n-k}\right)^{n-k+1/2} \;=\; n^k\left(1+\frac{k}{n-k}\right)^{n-k+1/2}.$$

We need a result from calculus:

$$\text{If } x \text{ is small, then } \ln(1+x) \;=\; x - x^2/2 + O(x^3) \text{ and so } 1 + x \;=\; \exp(x - x^2/2 + O(x^3)). \qquad 1.1$$

If you know Taylor's Theorem, you should be able to prove it; otherwise, just accept the result. Since k is much smaller than n, $\frac{k}{n-k}$ is small. Let it be x. By (1.1),

$$\left(1+\frac{k}{n-k}\right)^{n-k+1/2} \;=\; \exp(A(n-k+1/2)) \qquad \text{where} \qquad A \;=\; \frac{k}{n-k} - \frac{\left(\frac{k}{n-k}\right)^2}{2} + O((k/(n-k))^3).$$

With some algebra and the ability to work with $O(\,)$, one can deduce that

$$\exp(A(n-k+1/2)) \;=\; \exp(k - k^2/2n + O(k^3/n^2)).$$

[1] The notation in the next equations is discussed in Appendix B. It simply means that $n! \,/\, \left(\sqrt{2\pi n}(n/e)^n\right) \to 0$ as $n \to \infty$.

These manipulations are beyond what we expect of you at this point, so we'll omit them—you'll have to figure out how to do them or just accept this result.

Putting all this together:

$$\frac{n!}{(n-k)!} \sim \frac{n^{n+1/2}e^{-k}}{(n-k)^{n-k+1/2}} = n^k e^{-k} \exp(k + k^2/2n + O(k^3/n^2)).$$

If $k^3 = o(n^2)$, then $O(k^3/n^2) = o(1)$ and so $\exp(O(k^3/n^2)) = e^{o(1)} \sim 1$. Thus we have

$$\frac{n!}{(n-k)!} \sim n^k e^{-k^2/2n} \qquad \text{provided} \qquad k = o(n^{2/3}). \tag{1.2}$$

For example, by Theorem 1.4, the number of 200-lists without repeats that can be formed from a 10,000-set is about $10^{800}/e^2$. □

Example 1.11 Words from a collection of letters How many "words" of length k can be formed from the letters in ERROR when no letter may be used more often than it appears in ERROR? (A "word" is any list of letters, pronounceable or not.) If you are familiar with the game of Scrabble®, you can imagine that you have 5 tiles, namely one E, one O, and three R's. We cannot use 5^k since unlimited repetition is not allowed. On the other hand, we cannot use $(5)_k$ since repetition is allowed. At present, all we can do is carefully list the possibilities. Here they are in alphabetical order.

$k = 1$: E, O, R

$k = 2$: EO, ER, OE, OR, RE, RO, RR

$k = 3$: EOR, ERO, ERR, OER, ORE, ORR, REO, RER, ROE, ROR, RRE, RRO, RRR

$k = 4$: EORR, EROR, ERRO, ERRR, OERR, ORER, ORRE, ORRR, REOR, RERO, RERR, ROER, RORE, RORR, RREO, RRER, RROE, RROR, RRRE, RRRO

$k = 5$: EORRR, ERORR, ERROR, ERRRO, OERRR, ORERR, ORRER, ORRRE, REORR, REROR, RERRO, ROERR, RORER, RORRE, RREOR, RRERO, RROER, RRORE, RRREO, RRROE

This is obviously a tedious process—try it with ERRONEOUSNESS. We will explore better methods in Examples 1.19, 3.3 (p. 69), and 11.6 (p. 319). □

Example 1.12 Circular arrangements How many ways can n people be seated on a Ferris wheel with exactly one person in each seat? Equivalently, we can think of this as seating the people at a circular table with n chairs. Two seatings are defined to be "the same" if one can be obtained from the other by rotating the Ferris wheel (or rotating the seats around the table).

If the people were seated in a straight line instead of in a circle, the answer would be $n!$. Can we convert the circular seating into a linear seating (i.e., an ordered list)? In other words, *can we convert the unsolved problem to a solved one*? Certainly—simply cut the circular arrangement between two people and unroll it. Thus, to arrange n people in a linear ordering,

first arrange them in a circle AND then cut the circle.

According to our AND/OR technique, we must prove that each linear arrangement arises in *exactly one way* with this process.

- Since a linear seating can be rolled up into a circular seating, it can also be obtained by unrolling that circular seating. Hence each linear seating arises *at least once*.
- Since the people at the circular table are all different, the place we cut the circle determines who the first person in the linear seating is, so each cutting of a circular seating gives a different linear seating. Obviously two different circular seatings cannot give the same linear seating. Hence each linear seating arises *at most once*.

1				1		111112	112111
1	1	111111		2	1	111121	121111
1	1			1	1		
1				1		111211	211111
1		112112		1		112121	121112
2	1	121121		1	1	121211	211121
1	2			2	2		
1		211211		1		212111	111212
1				1		112221	221112
2	2	121212		1	1	122211	211122
1	1	212121		2	2		
2				2		222111	111222

Figure 1.1 Some circular arrangements with the corresponding linear arrangements.

Putting these two observations together, we see that each linear seating arises *exactly once*. By the Rule of Product,

$$\begin{aligned} n! \ &= (\text{number of circular arrangements}) \times \\ &\quad (\text{number of places to cut the circle}). \end{aligned}$$

Hence the number of circular arrangements is $n!/n = (n-1)!$.

Our argument was somewhat indirect. We can derive the result by a more direct argument. For convenience, let the people be called 1 through n. We can read off the people in the circular list starting with person 1. This gives a linear ordering of $\underline{n}$ that starts with 1. Conversely, each such linear ordering gives rise to a circular ordering. Thus the number of circular orderings equals the number of such linear orderings. Having listed person 1, there are $(n-1)!$ ways to list the remaining $n-1$ people. Thus the number of circular arrangements is $(n-1)!$.

If we are making circular necklaces using n distinct beads, then the arguments we have just given prove that there are $(n-1)!$ possible necklaces provided we are not allowed to flip necklaces over. What happens if the beads are not distinct?

The direct method fails if there are multiple copies of bead 1 because we don't know where to start reading. What about the indirect method? The different cuttings of the circular arrangement may not be distinct. Let's have a look at an example to see why. We'll take a circular arrangement with six "places" and put beads of type 1 and 2 around the circle, where we can use any number of each of the two types of beads. In Figure 1.1 are some distinct necklaces and, next to each, the distinct linear arrangements we get by unrolling. There are 2^6 different linear arrangements. Since some necklaces have less than six unrollings, $2^6/6$ is an underestimate of the number of necklaces.

We can describe what we're doing as follows: Call two lists (i.e., linear arrangements) "equivalent" if one can be gotten from the other by "circularly permuting" the elements; that is, by shifting everything down some fixed number of positions and putting what is shifted off the end at the beginning. The lists fall into sets of equivalent lists, each set corresponding to one circular seating. Figure 1.1 can be thought of as containing six such sets of equivalent lists. The number of necklaces is the number of sets of equivalent lists.

Although we will not study tools for dealing with problems having equivalences until Chapter 4, there is one important class of problems with equivalences that we can deal with now. Suppose we allow the list entries to be rearranged in any fashion; in other words, we want to count unordered lists. We'll take up this subject in the next section. □

Our first derivation of the formula, $n!/n$, for seating n people at a circular table illustrates an important but obvious principle:

No matter how you count a set, the number is always the same.

For circular arrangements, we counted the set of linear arrangements in two ways. Another obvious principle is

If there is a one-to-one correspondence between two sets, then they are the same size.

This can be used to show that two counting problems have the same answer. In the next example we consider a famous example of this—the Catalan numbers, which arise in a variety of counting problems.

Example 1.13 Catalan numbers Suppose we have an election between two candidates and the ballots are counted one-by-one. Further suppose that the first candidate is never behind (she's always ahead or tied), but that the final count ends in a tie with each candidate getting n votes. How many ways can this happen? The answer is called the Catalan number C_n. We are looking at ordered lists of that contain n ones and n twos such that, for all k, the number of twos in the first k elements is at most $k/2$. The lists for $n \leq 3$ are

$$12 \quad 1122 \quad 1212 \quad 111222 \quad 112122 \quad 112212 \quad 121122 \quad 121212$$

and so $C_1 = 1$, $C_2 = 2$, $C_3 = 5$. In general $C_n = \frac{(2n)!}{n!\,(n+1)!}$. We won't derive the formula for C_n now, but we want to look at other problems that have the same answer. (If you look up Catalan numbers in the index, you can find a derivation of the formula in the text as well as other problems that have the same answer.)

In computer science we have the notion of a *stack*. This is an ordered list with two operations:

- PUSH: Add an item to the end of the list.
- POP: Remove an item from the end of the list.

It is illegal to attempt to "POP" an empty stack. How many ways can we start out with an empty stack, PUSH and POP in some order and end up with an empty stack at the end? There must be the same number of PUSHs and POPs. Suppose there are n of each. You should be able to convince yourself that this is the same as the election problem and so the answer is C_n.

Suppose we have n things we want to multiply together. In general, $ab \neq ba$ so order matters; however, we can group them in any way we want. (This is true if the things being multiplied are matrices.) For example, here are the ways we could group four things for multiplication.

$$a(b(cd)) \quad a((bc)d) \quad (ab)(cd) \quad (a(bc))d \quad ((ab)c)d.$$

We can do this with a stack using one of two operations:

- STORE: PUSH the next thing onto the stack
- MULT: POP two things off the stack and PUSH their product onto the stack.

For example, to do $a((bc)d)$ we would do

STORE, STORE, STORE, MULT, STORE, MULT, MULT.

There must be n STOREs to get all n items onto the stack. There must be $n-1$ MULTs. The number of STOREs in the first k things must exceed the number of MULTs. (Can you see why the last two statements are true?) Forgetting the first STORE, this is just the original voting problem with $n-1$ votes each. Thus the answer is C_{n-1}.

A regular n-gon can be cut up into triangles all of whose vertices are vertices of the n-gon. To do this, one must draw $n-3$ nonintersecting diagonals. We call this a "triangulation of the n-gon." Here are the five triangulations of the pentagon.

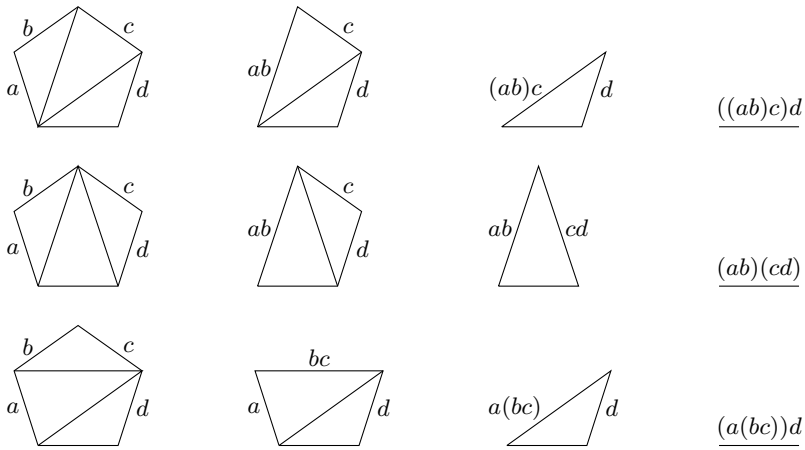


Figure 1.2 The reduction of three of the five triangulations of the pentagon to multiplications of *abcd*.

We want to know how many triangulations there are for a regular n-gon. This is trickier than the previous correspondences. First, we need to know a little about what the triangulations look like.

It turns out that, for $n > 3$, every triangulation has $n-3$ diagonals, $n-2$ triangles and exactly two triangles that contain two edges of the original n-gon. Actually, any of these three claims can be used to prove the other two. To see this, suppose there are D diagonals and T triangles. Then the triangles have a total of $3T$ edges. These edges come from the original n-gon and from *both sides* of the diagonals. Thus $3T = n + 2D$. It is clear that every triangle contains either one or two edges of the n-gon. Call the number of these triangles T_1 and T_2, respectively. Then $T_1 + T_2 = T$ and $T_1 + 2T_2 = n$. In summary

$$3T = n + 2D \qquad T_1 + T_2 = n \qquad T_1 + 2T_2 = n.$$

We have three equations in the four unknowns D, T, T_1 and T_2. If any of these is known (e.g., $D = n - 3$), we can solve the equations for the other three. Which value should we determine so that the others can be found?

We'll prove that $D = n - 3$. This is even true for 3-gons (triangles) since no diagonals are needed. We'll use induction for $n > 3$. Suppose we are given a triangulation of an n-gon. Cut it along any diagonal to split it into two polygons. Let the number of sides of the two polygons be k_1 and k_2. Since cutting along the diagonal has given us two new sides, $k_1 + k_2 = n + 2$. Notice that the k_1-gon and k_2-gon are triangulated. By induction, the k_1-gon has $k_1 - 3$ diagonals and the k_2-gon has $k_2 - 3$. Thus, counting the diagonal we cut along, the number of diagonals in the original n-gon triangulation is

$$(k_1 - 3) + (k_2 - 3) + 1 \;=\; (k_1 + k_2) - 5 \;=\; (n + 2) - 5 \;=\; n - 3,$$

and the induction is complete.

We'll now describe a method for associating a multiplication of $n-1$ things with a triangulation of an n-gon. Draw the n-gon with one side at the bottom. We'll call this side the "base". Label all the sides except the base. (See the left side of Figure 1.2.) There are two triangles that have two sides belonging to the n-gon. Thus there must be a triangle with two labeled sides. Remove the labeled sides and place the product of their labels on the third side. Repeat this process until we are left with a labeled base. Figure 1.2 contains examples.

To complete the process we need to know that this gives us a one-to-one correspondence between the triangulations and the multiplications. Simply write the multiplication on the base and reverse the steps. In other words, read Figure 1.2 from right to left instead of from left to right. We leave it

to you to convince yourself that every multiplication leads to a unique triangulation and vice versa. Thus there are C_{n-2} triangulations of a regular n-gon.

We have looked at only a few of the dozens of combinatorial interpretations of the Catalan numbers. □

Exercises

In each of the exercises, indicate how you are using the Rules of Sum and Product. It is instructive to first do these exercises using only the techniques introduced so far and then, after reading the next section, to return to these exercises and look for other ways of doing them. More generally, looking back at earlier sections to get a new viewpoint is often helpful. We do this in the text to some extent, but you should do it on your own, too.

1.2.1. Find to two decimal places the answer to the birthday question asked in Example 1 (p. 1).
Hint. Assigning birthdays to 30 people is the same as forming an ordered list of 30 dates.

1.2.2. Use (1.2) to estimate the solution to the birthday problem in Example 1 (p. 1).

1.2.3. How many ways are there to form an ordered list of two distinct letters from the set of letters in the word COMBINATORICS? three distinct letters? four distinct letters?

1.2.4. Repeat the previous problem when the letters need not be distinct but cannot be used more often than they appear in COMBINATORICS.

1.2.5. We are interested in forming 3 letter words ("3-words") using the letters in LITTLEST. For the purposes of the problem, a "word" is any ordered list of letters.

(a) How many words can be made with no repeated letters?

(b) How many words can be made with unlimited repetition allowed?

(c) How many words can be made if repeats are allowed but no letter can be used more often than it appears in LITTLEST?

1.2.6. Redo the previous exercise for k-words. The last part should be starred. It can be done if you treat each value of $k \leq 8$ separately and carefully break it down into cases with OR. Even so, you should study the next section before you attempt it.

1.2.7. Each of the following belongs to one of the four types of things described in Example 1.13. In each case, list the other three things that correspond to it using the correspondences in the example.

(a) 1122112122

(b) $(a(bc))(((de)f)g)$

(c)

1.2.8. Suppose we have an election as in Example 1.13, but now the first candidate is *always* ahead except for the 0–0 and n–n ties at the start and finish. How many ways can this happen?

1.2.9. By 2001 spelling has deteriorated considerably. The dictionary defines the spelling of "relief" to be any combination (with repetition allowed) of the letters R, L, F, I and E subject to certain constraints listed below How many spellings are possible? The most popular spelling is the one that, in dictionary order, is five before the spelling RELIEF. What is it?

(i) The number of letters must not exceed 6.

(ii) The word must contain at least one L.

(iii) The word must begin with an R and end with an F.

(iv) There is just one R and one F.

1.2.10. By the year 2010, further deterioration in spelling has relaxed the last condition listed above so that we can have any number of initial R's and any number of terminal F's, provided there is at least one of each. How many spellings are possible? Which spelling is five before RELIEF in dictionary order?

1.2.11. Prove that the number of ordered lists without repeats that can be constructed from an n-set is very nearly $n!e$. The lists can be of any length.
Hint. Recall that from Taylor's Theorem in calculus $e^x = 1 + x + x^2/2! + x^3/3! + \cdots$.

1.2.12. In this exercise, we look at ways of seating n people at a long table that has n seats. In (c)–(e), n is even.
Hint. If you fix a corner of the table and read out the seating arrangement counterclockwise starting at that corner, you have an ordered list. If you draw pictures, you should be able to see how many ordered lists give an equivalent seating arrangement; for example, by reversing right and left in (b).

(a) Suppose that everyone is to be seated on one side of the table. How many ways can it be done?

(b) Suppose we don't care if left and right are interchanged; that is, seating $A, B, C, \ldots$ from left to right will be considered the same as doing it from right to left. (This is reasonable if all we care about is who a person's neighbors are.) How many ways can this be done?

(c) How many ways can it be done if n is even and half the people are seated on each side of the table? Assume that we can tell the two sides of the table apart; for example, one side faces a wall and the other side faces into the room. Also assume seating left to right is different from seating right to left.

(d) Suppose we seat people on both sides as in (c) and all we care about is who a person's neighbors are on each side, as in (b).

(e) Suppose we are dealing with a seating as in (d), but now we also care about who is sitting opposite a person as well as who a persons neighbors on each side are.

*1.2.13. This exercise contains several related questions. In each case we would like a formula that answers the question "How many ways can p people run for k offices?" under the given constraints. Unless the constraints say otherwise, a person may run for no offices. At present, we have the tools to do only two parts of this exercise. The challenge in this exercise is to avoid finding wrong "solutions" to the parts that we are unable to do, as well as doing the two parts we can do now. One way you can check your "solution" is to actually list all the possible ways p people can run for k offices for each of the parts for some small values of p and k. We will return to this exercise later as we develop tools for doing other parts of it.

(a) Each person must be a candidate for at most one office.

(b) Each person must be a candidate for exactly one office and each office must have at least one candidate.

(c) Each person must be a candidate for at most one office and each office must have at least one candidate.

(d) Each person can be a candidate for any number of offices (including none) and each office must have at least one candidate.

(e) Each person must be a candidate for at least one office and each office must have at least one candidate.

*1.2.14. In Example 1.12: How many are there of length 3 made from A's and B's? Length 5? Can you prove a general result for all primes? What about allowing more than two kinds of letters?

1.3 Sets

People use $C(n,k)$ to stand for the number of different k-subsets that can be formed from an n-set. The notation $\binom{n}{k}$ is also frequently used. These are called *binomial coefficients* and are read "n choose k." Think about how you might count k-subsets, that is, unordered k-lists.

* * * Stop and think about this! * * *

You may have concluded that this seems a bit trickier to do than counting ordered lists. Can we rephrase the problem in a way that lets us solve it, or convert it to an ordered list problem?

- An unordered k-list of distinct elements from a set S is simply a k-subset of S. This doesn't seem to be of any help at present; however, we will generally think in terms of subsets rather than unordered lists since the subset view is used more often in the literature.
- If the original set consisted of something ordered, like the integers, we could introduce a "natural" ordering to an unordered list, namely the one in which the elements are in increasing order (or, if you prefer, decreasing order). Again this doesn't seem to help, but provides a possibly useful interpretation.
- We can adjust the previous idea a bit. Let's consider all possible orderings of our lists. This is a way of constructing all ordered lists with distinct elements in two steps: First construct an unordered list with no repeats, then order it. An unordered k-list with no repeats is simply a k-set. We can order it by forming a k-list without repeats from it. By Theorem 1.4 (p. 11), we know that this can be done in $k!$ ways. By the Rule of Product, there are $C(n,k)k!$ ordered k-lists with no repeats. By Theorem 1.4 again, this number is $n(n-1)\cdots(n-k+1) = n!/(n-k)!$. Dividing by $k!$, we have

Theorem 1.6 Binomial coefficient formula *The value of the binomial coefficients is*

$$\binom{n}{k} = C(n,k) = \frac{n(n-1)\cdots(n-k+1)}{k!} = \frac{n!}{k!\,(n-k)!}.$$

Example 1.14 A generating function for binomial coefficients We'll now approach the problem of evaluating $C(n,k)$ in another way. In other words, we'll "forget" the formula we just derived and start over with a new approach.

You may ask "Why waste time using another approach when we've already gotten what we want?" We gave a partial answer to this earlier. Here is a more complete response.

- By looking at a problem from different viewpoints, we may come to understand it better and so be more comfortable working similar problems in the future.
- By looking at a problem from different viewpoints, we may discover that things we previously thought were unrelated have interesting connections. These connections might open up easier ways to solve some types of problems and may make it possible for us to solve problems we couldn't do before.
- A different point of view may lead us to a whole new approach to problems, putting powerful new tools at our disposal.

In the approach we are about to take, we'll begin to see a powerful tool for solving counting problems. It's called "generating functions" and it lets us put calculus and related subjects to work in combinatorics. In later chapters, we'll devote more time to generating functions. Now, we'll just get a brief glimpse of them.

Suppose that $S = \{x_1, \ldots, x_n\}$ where x_1, x_2, ... and x_n are variables as in high school algebra. Let $P(S) = (1+x_1)\cdots(1+x_n)$. The first three values of $P(S)$ are

$$n = 1: \quad 1 + x_1$$
$$n = 2: \quad 1 + x_1 + x_2 + x_1x_2$$
$$n = 3: \quad 1 + x_1 + x_2 + x_3 + x_1x_2 + x_1x_3 + x_2x_3 + x_1x_2x_3.$$

From this you should be able to convince yourself that $P(S)$ consists of a sum of terms where each term represents one of the subsets of S as a product of its elements. Can we reach some understanding of why this is so? Yes, but we'll only explore it briefly now. The understanding relates to the Rules of Sum and Product. Interpret plus as OR, times as AND and 1 as "nothing." Then $(1 + x_1)(1 + x_2)(1 + x_3)$ can be read as

- include the factor 1 in the term OR include the factor x_1 AND
- include the factor 1 in the term OR include the factor x_2 AND
- include the factor 1 in the term OR include the factor x_3.

This is simply a description of how to form an arbitrary subset of $\{x_1, x_2, x_3\}$. On the other hand we can form an arbitrary subset by the rule

- Include nothing in the subset OR
- include x_1 in the subset OR
- include x_2 in the subset OR
- include x_3 in the subset OR
- include x_1 AND x_2 in the subset OR
- include x_1 AND x_3 in the subset OR
- include x_2 AND x_3 in the subset OR
- include x_1 AND x_2 AND x_3 in the subset.

If we drop the subscripts on the x_i's, then a product representing a k-subset becomes x^k. We get one such term for each subset and so it follows that the coefficient of x^k in the polynomial $f(x) = (1 + x)^n$ is $C(n, k)$; that is,

$$(1 + x)^n = \sum_{k=0}^{n} C(n, k)x^k. \tag{1.3}$$

Can this help us evaluate $C(n, k)$? Calculus comes to the rescue! Remember Taylor's Theorem? It tells us that the coefficient of x^k in $f(x)$ is $f^{(k)}(0)/k!$. Let $f(x) = (1 + x)^n$. You should be able to prove by induction on k that

$$f^{(k)}(x) = n(n - 1) \cdots (n - k + 1)\ (1 + x)^{n-k}.$$

Thus $c(n, k)$, the coefficient of x^k in $(1 + x)^n$, is

$$C(n, k) = \frac{f^{(k)}(0)}{k!} = \frac{n(n - 1) \cdots (n - k + 1)}{k!}.$$

We conclude this example with a useful formula that follows from (1.3). Since $(x + y)^n = x^n(1 + (y/x))^n$, it follows that the coefficient of $x^n(y/x)^k$ in $(x + y)^n$ is $C(n, k)$. This gives us the

Theorem 1.7 Binomial Theorem

$$(x + y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k}y^k.$$

The expressions we've been studying are called *generating functions.* □

Example 1.15 Card hands: Full house Card hands provide a source of some simple sounding but tricky set counting problems. A standard deck of cards contains 52 cards, each of which is marked with two labels. The first label, called the "suit," belongs to the set

$$\{\clubsuit, \heartsuit, \diamondsuit, \spadesuit\}.$$

The second label, called the "value" belongs to the set

$$\{2, 3, 4, 5, 6, 7, 8, 9, 10, \mathrm{J}, \mathrm{Q}, \mathrm{K}, \mathrm{A}\}.$$

Each pair of labels occurs exactly once in the deck. A hand is a subset of a deck. Two cards are a pair if they have the same values.

How many 5 card hands consist of a pair and a triple? (In poker, such a hand is called a full house.)

To calculate this we describe how to construct such a hand:

- Choose the value for the pair AND
- Choose the value for the triple different from the pair AND
- Choose the 2 suits for the pair AND
- Choose the 3 suits for the triple.

This produces each full house exactly once, so the number is the product of the answers for the four steps, namely

$$13 \times 12 \times C(4,2) \times C(4,3) \;=\; 3{,}744.$$

What is the probability of being dealt a full house? There are $\binom{52}{5}$ distinct hands of cards so we could simply divide the previous answer by this number. This approach looks at the *result* of the deal rather than the actual deal. Why do we say that? When a hand of cards is dealt, the *order* in which you receive the cards matters. Thus:

- If we look at the resulting hand, then the order of the cards doesn't matter. That's the way we just got the answer.
- If we look at the dealing process, then the order of the cards matters. We'll do the problem that way next.

Each of the $52 \times 51 \times 50 \times 49 \times 48$ ways of dealing five cards from 52 as equally likely. Then we should divide this into the number of ways of being dealt a full house. Since all the cards in a hand of five cards are different, they can be ordered in 5! ways. Hence the probability of being dealt a full house is $\frac{3{,}774 \times 5!}{52 \times 51 \times 50 \times 49 \times 48}$, which gives the same answer as before, $3{,}744/\binom{52}{5}$.

Let's phrase these in terms of probability spaces. We'll use the uniform distribution on both spaces.

- Resulting hand: The space contains all $\binom{52}{5}$ 5-card subsets of the 52-card deck.
- Dealing process: The space contains all $52 \times 51 \times 50 \times 49 \times 48$ 5-card lists without repeats that can be made from a 52-card deck.

Which approach should you use? That's up to you. However, whichever approach you choose, there you may have problems if there is more than one copy of a card. For example, one might add two jokers to a deck or one might combine two identical decks as in canasta. In this case, it's probably easiest and safest to pretend that the cards have been marked so you can tell them apart; for example, call them joker-1 and joker-2. □

Example 1.16 Card hands: Two pairs We'll continue with our poker hands. How many 5 card hands consist of two pairs? A description of a hand always means that there is nothing better in the hand, so "two pairs" means we don't have a full house or four of a kind.

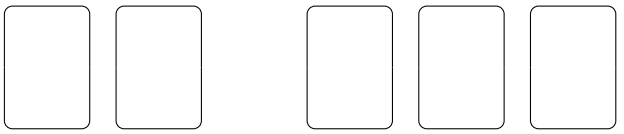
One thing we might try is to go back to the preceding example's description of how to construct a full house and two simple changes: (a) replace "triple" by "second pair" and (b) add a choice for the card that belongs to no pair. This is wrong! Each hand is constructed *twice*, depending on which pair is the "second pair." Try it! What happened? Before choosing the cards for a pair and a triple, we can distinguish the pair from the triple because one contains two cards and the other contains three. We can't distinguish the two pairs, though, until the values are specified. This is an example of a situation where we can easily make mistakes if we forget that "AND" means "AND then." Here's a correct description, with "then" put in for emphasis.

- Choose the values for the two pairs AND then
- Choose the 2 suits for the pair with the larger value AND then
- Choose the 2 suits for the pair with the smaller value AND then
- Choose the remaining card from the 4×11 cards that have different values than the pairs.

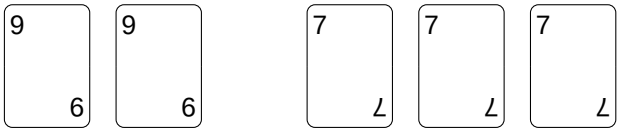
The value is

$$\binom{13}{2} \times \binom{4}{2} \times \binom{4}{2} \times 44 \;=\; 123{,}552.$$

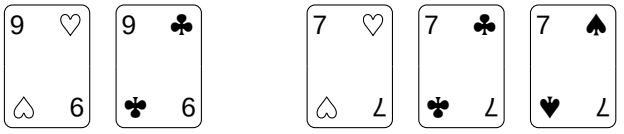
You may find what we've just been through disquieting: How can you decide between distinguishable and indistinguishable? The answer is simple: Draw a picture of the cards and fill in the information after each step. Let's do this for the full house and the two pair problems. To begin with, we have five blank cards. For the full house, we divide the cards up into a pair and a triple:



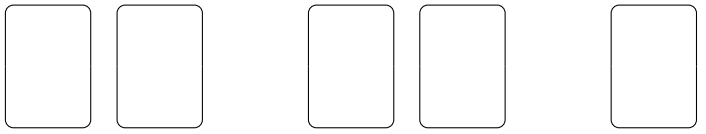
We can tell the two groups apart, so it makes sense to talk about assigning a value to the pair, say 9, and to the triple, say 7, to obtain



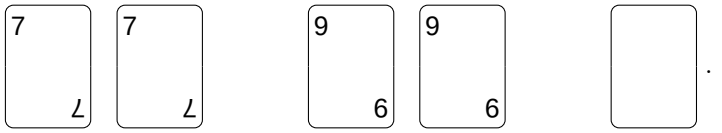
We can't tell the two nines apart, so all we can do is choose a subset of two suits to assign to them; likewise for the triple. We might choose $\{\heartsuit, \clubsuit\}$ and $\{\heartsuit, \spadesuit, \clubsuit\}$ and obtain



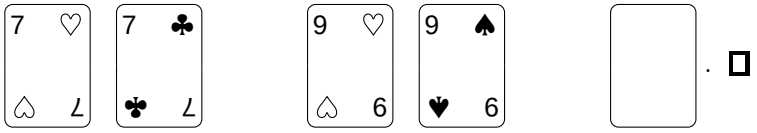
Now look at the case of two pairs. We have



Since we can't distinguish the two pairs, all we can do is choose a set of two values, say $\{7, 9\}$ and put them on the cards:



Now we can distinguish between the pairs. For the pair of sevens we might choose the set $\{\heartsuit, \clubsuit\}$ of suits, and for the nines, $\{\heartsuit, \spadesuit\}$. As a result, we have



Example 1.17 Smorgasbord College programs Smorgasbord College allows students to study in three principal areas: (a) Swiss naval history, (b) elementary theory and (c) computer science. The number of upper division courses offered in these fields are 2, 92, and 15 respectively. To graduate a student must choose a major and take 6 upper division courses in it, and also choose a minor and take 2 upper division courses in it. Swiss naval history cannot be a major because only 2 upper division courses are offered in it.

How many programs are possible?

The possible major-minor pairs are b-a, b-c, c-a, and c-b. By the Rule of Sum we can simply add up the number of programs in each combination. Those programs can be found by the Rule of Product. The number of major programs in (b) is $C(92, 6)$ and in (c) is $C(15, 6)$. For minor programs: (a) is $C(2, 2) = 1$, (b) is $C(92, 2) = 4186$ and (c) is $C(15, 2) = 105$. Since the possible programs are constructed by

$$\begin{aligned} &\Big(\text{major (b) AND } \Big(\text{minor (a) OR minor (c)}\Big)\Big) \\ \text{OR } &\Big(\text{major (c) AND } \Big(\text{minor (a) OR minor (b)}\Big)\Big), \end{aligned}$$

the number of possible programs is

$$\binom{92}{6}(1 + 105) + \binom{15}{6}(1 + 4186) \;=\; 75{,}606{,}201{,}671,$$

a rather large number. □

Example 1.18 Multinomial coefficients Suppose we are given k boxes labeled 1 through k and a set S and are told to distribute the elements of S among the boxes so that the ith box contains exactly m_i elements. How many ways can this be done?

Let $n = |S|$. Unless $m_1 + \cdots + m_k = n$, the answer is zero because we don't have the right number of objects. Therefore, we assume from now on that

$$m_1 + \cdots + m_k = n.$$

Here's a way to describe filling the boxes.

- Fill the first box (There are $C(n, m_1)$ ways.) AND
- Fill the second box (There are $C(n - m_1, m_2)$ ways.) AND

  . . . . . .
- Fill the kth box. (There are $C(n - (m_1 + \ldots + m_{k-1}), m_k) = C(m_k, m_k) = 1$ ways.)

Now apply the Rule of Product, use the formula $C(p,q) = p!/q!(p-q)!$ everywhere, and cancel common factors in numerator and denominator to obtain $n!/m_1!m_2!\cdots m_k!$. This is called a *multinomial coefficient* and is written

$$\binom{n}{m_1, m_2, \ldots, m_k} = \frac{n!}{m_1!m_2!\cdots m_k!}, \tag{1.4}$$

where $n = m_1 + m_2 + \ldots + m_k$. In multinomial notation, the binomial coefficient $\binom{n}{k}$ would be written $\binom{n}{k,(n-k)}$. You can think of the first box as the k things that are chosen and the second box as the $n-k$ things that are not chosen.

Before you read on, try to think of an ordered list interpretation for the multinomial coefficient.

* * * Stop and think about this! * * *

Think of the objects being distributed as positions in a word and the boxes as letters. If the object "position 3" is placed in the box "D," then the letter D is the third letter in the word. The multinomial coefficient is then the number of words that can be made so that letter i appears exactly m_i times. A word can be thought of as an ordered list of its letters. □

Example 1.19 Words from a collection of letters Using the idea at the end of the previous example, we can more easily count the words that can be made from ERROR, a problem discussed in Example 1.11 (p. 13). Suppose we want to make words of length k. Let m_1 be the number of E's, m_2 the number of O's and m_3 the number of R's. By considering all possible cases for the number of each letter, you should be able to see that the answer is the sum of $\binom{k}{m_1,m_2,m_3}$ over all m_1, m_2, m_3 such that

$$m_1 + m_2 + m_3 = k, \quad 0 \le m_1 \le 1, \quad 0 \le m_2 \le 1, \quad 0 \le m_3 \le 3.$$

Thus we obtain

$$\begin{aligned}
k=1: &\quad \binom{1}{0,0,1} + \binom{1}{0,1,0} + \binom{1}{1,0,0} = 3\\
k=2: &\quad \binom{2}{0,0,2} + \binom{2}{0,1,1} + \binom{2}{1,0,1} + \binom{2}{1,1,0} = 7\\
k=3: &\quad \binom{3}{0,0,3} + \binom{3}{0,1,2} + \binom{3}{1,0,2} + \binom{3}{1,1,1} = 13\\
k=4: &\quad \binom{4}{0,1,3} + \binom{4}{1,0,3} + \binom{4}{1,1,2} = 20\\
k=5: &\quad \binom{5}{1,1,3} = 20.
\end{aligned}$$

This is better than in Example 1.11. Instead of having to list words, we have to list triples of numbers and each triple generally corresponds to more than one word. Here's the lists for the preceding computations

$$\begin{aligned}
k=1: &\quad 0,0,1 \;\; 0,1,0 \;\; 1,0,0\\
k=2: &\quad 0,0,2 \;\; 0,1,1 \;\; 1,0,1 \;\; 1,1,0\\
k=3: &\quad 0,0,3 \;\; 0,1,2 \;\; 1,0,2 \;\; 1,1,1\\
k=4: &\quad 0,1,3 \;\; 1,0,3 \;\; 1,1,2\\
k=5: &\quad 1,1,3
\end{aligned}$$

In Example 3.3 (p. 69), we will see how to do this more systematically and efficiently. □

Example 1.20 Card hands and multinomial coefficients We'll redo Examples 1.15 and 1.16, and then discuss the general situation using multinomial coefficients.

To form a full house, we must choose a face value for the triple, choose a face value for the pair, and leave eleven face values unused. This can be done in $\binom{13}{1,1,11}$ ways. We then choose the suits for the triple in $\binom{4}{3}$ ways and the suits for the pair in $\binom{4}{2}$ ways.

To form two pair, we must choose two face values for the pairs, choose a face value for the single card, and leave ten face values unused. This can be done in $\binom{13}{2,1,10}$ ways. We then choose suits for each of the face values in turn, so we must multiply by $\binom{4}{2}\binom{4}{2}\binom{4}{1}$.

Imagine an eleven card hand containing two triples, a pair and three single cards. You should be able to see that the number of ways to do this is

$$\binom{13}{2,1,3,7}\binom{4}{3}\binom{4}{3}\binom{4}{2}\binom{4}{1}\binom{4}{1}\binom{4}{1}.$$

Let's do the general case. Suppose our hand must contain c_1 singles, c_2 pairs, c_3 triples and c_4 four-of-a-kinds. The number of such hands is

$$\binom{13}{c_1,c_2,c_3,c_4,k}\binom{4}{1}^{c_1}\binom{4}{2}^{c_2}\binom{4}{3}^{c_3}\binom{4}{4}^{c_4},$$

where $k = 13 - c_1 - c_2 - c_3 - c_4$ is the number of face values not in the hand. □

Example 1.21 Choosing Teams Given 22 people, how many ways can we divide them into 4 teams of 5 players each plus 2 referees? If the teams and referees were labeled, the answer would be $\binom{22}{5,5,5,5,1,1}$. Given 4 different teams and two referees, there are 4! ways to label the teams as Team 1, 2, 3, and 4, and there are 2 ways to label the referees, so the answer is

$$\binom{22}{5,5,5,5,1,1}\frac{1}{4!\times 2} = \frac{(22)!}{2(5!)^4\,4!}.$$

Suppose now we must divide up the teams into pairs that compete against each other, and we assign a referee to each pair. If they were called Match #1 and Match #2, we could fill out Match #1 by choosing 2 of the 4 teams and 1 of the referees. Those left are Match #2. This gives us $\binom{4}{2}\times 2 = 12$. Thus we have $\frac{(22)!}{2(5!)^4\,4!}\times 12$. Of course, there isn't really a Match #1 and Match #2, but there are two ways to assign match labels and so we must divide the answer we just got by 2. □

***Example 1.22 Incomparable sets** Let A be an n-set. By a *Sperner family* on A we mean a family of subsets of A such that no subset in the family is contained in any other subset in the family. For example, let $A = \{1,2,3,4,5\}$. Then

$$\{1,2,4\} \quad \{1,5\} \quad \{2,4,5\} \quad \{3,5\}$$

is a Sperner family but

$$\{1,2,4\} \quad \{1,5\} \quad \{2,4\} \quad \{3,5\}$$

is not.

What is the largest number of subsets that we can have in a Sperner family of an n-set?

Clearly the family of all k-subsets of A is a Sperner family. Thus we can construct Sperner families of size at least $\binom{n}{k}$. What value of k will make this as large as possible? One way to find the value of k is to look at the ratio of $\binom{n}{k}$ to $\binom{n}{k-1}$. When this ratio exceeds 1, the sequence of binomial coefficients is increasing and when it is less than 1 the sequence is decreasing. Since

$$\frac{\binom{n}{k}}{\binom{n}{k-1}} = \frac{n!\,(n-k+1)!\,(k-1)!}{(n-k)!\,k!\,n!} = \frac{n-k+1}{k} = \frac{n+1}{k} - 1,$$

we see that the sequence is increasing when $(n+1)/k > 2$ and is decreasing when $(n+1)/k < 2$. It follows that

$$\binom{n}{k} \text{ is a maximum at } \begin{cases} k = n/2, \text{ when } n \text{ is even;} \\ k = (n-1)/2 \text{ and } k = (n+1)/2, \text{ when } n \text{ is odd.} \end{cases}$$

$\lfloor x \rfloor$, the floor function, denotes the largest integer not exceeding x. With this, we can write our conclusions in the form: There is a Sperner family of size $\binom{n}{\lfloor n/2 \rfloor}$, which can be obtained by taking all $\lfloor n/2 \rfloor$-subsets of A.

Sperner proved that this result is best possible: there are no larger Sperner families. We now present an adaptation of Lubell's proof of this result.

Call a k-set B an "initial part" of a list L if the first k elements of L are the elements of B. Let $\mathcal{S}$ be a Sperner family on the n-set A. Consider an n-list L of the n elements of A. We claim that at most one set in $\mathcal{S}$ can be an initial part of L, for if there were two such sets, one would correspond to a longer initial part than the other and so contain the other as a subset.

On the other hand, a k-set B is the initial part of exactly $k!\,(n-k)!$ n-lists. Why is this? The first k elements of the list must be some arrangement of the elements of B, AND the remaining $n-k$ elements of the list must be some arrangement of the remaining $n-k$ elements of S. Furthermore, any list satisfying these conditions has B as an initial part. By the Rule of Product and Theorem 1.4 (p. 11), there are $k!\,(n-k)!$ permutations which have B as an initial part. Adding this up over all B in $\mathcal{S}$, we obtain the number of rearrangements of A that have sets in $\mathcal{S}$ as initial parts. (This uses the result from previous paragraph that each list has at most one element of $\mathcal{S}$ as an initial part.) Since there are $n!$ lists, we have proved

$$\sum_{B \in \mathcal{S}} |B|!\,(n - |B|)! \;\leq\; n!.$$

Dividing by $n!$ we obtain

$$\sum_{B \in \mathcal{S}} \frac{1}{\binom{n}{|B|}} \;\leq\; 1. \tag{1.5}$$

This inequality is the key to the proof. By our earlier work on the size of binomial coefficients, we know that each term in the sum in (1.5) is at least as big as $1/\binom{n}{\lfloor n/2 \rfloor}$. Consequently, the sum in (1.5) can have at most $\binom{n}{\lfloor n/2 \rfloor}$ terms. In other words the size of the Sperner family is at most $\binom{n}{\lfloor n/2 \rfloor}$—but we have already constructed Sperner families this big! This completes the proof. □

*Example 1.23 When are two subsets disjoint? Alice chooses a k-subset at random from and n-set. Bob chooses an l-subset at random from the same n-set. Find an exact expression and a simple estimate the probability that the two subsets are disjoint. For the estimate, you may assume that $k = o(n^{2/3})$ and $l = o(n^{2/3})$.

Call the probability $P(n,k,l)$. By the Rule of Product, there are $\binom{n}{k}\binom{n}{l}$ ways to choose two subsets of the given sizes. There are $\binom{n}{k}\binom{n-k}{l}$ ways to choose two disjoint subsets of the given sizes. Since things are done at random, all choices are equally likely and so

$$P(n,k,l) \;=\; \frac{\binom{n}{k}\binom{n-k}{l}}{\binom{n}{k}\binom{n}{l}} \;=\; \frac{(n-k)!/(n-k-l)!}{n!/(n-l)!} \;=\; \frac{(n-k)!}{n!} \times \frac{(n-l)!}{((n-l)-k)!}.$$

This is the exact answer written in various forms.

The exact answer does not give us a good idea of how the probability behaves when the numbers are large. To get a simple estimate, we use (1.2):

$$\frac{n!}{(n-k)!} \;\sim\; n^k e^{-k^2/2n} \qquad \text{and} \qquad \frac{(n-l)!}{((n-l)-k)!} \;\sim\; (n-l)^k e^{-k^2/2(n-l)}.$$

Thus

$$P(n,k,l) \sim \frac{(n-l)^k e^{-k^2/2(n-l)}}{n^k e^{-k^2/2n}} = \left(1-\frac{l}{n}\right)^k \exp\left(-\frac{k^2 l}{2n(n-l)}\right).$$

We need to look at the two factors on the right. Inside the exponential we have $\frac{-k^2 l}{2n(n-l)}$. Since l is small compared to n, this is nearly $\frac{-k^2 l}{2n^2}$. Since $k = o(n^{2/3})$ and $l = o(n^{2/3})$, we have $k^2 l = o((n^{2/3})^2 n^{2/3})$. Combining the exponents on the right, $k^2 l = o(n^2)$. Thus $\frac{k^l}{2n(n-l)} \to 0 = o(1)$. Since the exponential of a number close to zero is close to one, (1.5) becomes

$$\begin{aligned} P(n,k,l) &\sim \left(1-\frac{l}{n}\right)^k = \exp\left(k\ln\left(1-\frac{l}{n}\right)\right) \\ &\sim \exp\big(k\big((-l/n)-(-l/n)^2/2+O(l^3/n^3)\big)\big) && \text{by (1.1)} \\ &= \exp\big(-kl/n-(kl^2/2n^2)+O(kl^3/n^3)\big). \end{aligned}$$

You should be able show that $kl^2/2n^2 = o(1)$ and $kl^3/n^3 = o(n^{-1/3})$. Thus we have

$$P(n,k,l) \sim e^{-kl/n} \quad \text{provided } k = o(n^{2/3}) \text{ and } l = o(n^{2/3}).$$

Our constraints on the growth of k and l was necessary so that we could obtain our result, but looking at the result we can see some justification for the constraints: When kl is much larger than n, the probability will be very close to 0 and so may be uninteresting. If k and l are about the same size, the low probability occurs when they grow faster than $n^{1/2}$. □

*Error Correcting Codes

We want to represent information by n-strings of zeroes and ones. For example, the information may be a letter of the alphabet. ASCII provides a way of doing this: an 8-string is used to represent the upper and lower case alphabet, the digits, the punctuation marks and some special "characters."

The ASCII representation of characters is quite sensitive to errors: if even a single entry in the 8-string is changed, we end up with a completely different character. This may be unacceptable. For example, suppose the characters are being transmitted over a data link which may have a small amount of static, the effect of which is to sometimes change a zero to a one or vice versa. A Soviet space probe was lost in 1989 because of a single character error in a lengthy control signal.

What can we do about the problem of errors in transmission?

One solution is to transmit the ASCII representation of each character some number $k > 1$ times. If $k = 2$ and the two transmitted values agree, we very likely have the correct value. If they disagree, we must ask the sender to try again. If $k = 3$ and the three transmitted values disagree, instead of asking for a retransmission, we can try to guess the answer by using a majority vote. For example, suppose we transmit three copies of 01010101, which we receive as 01010001, 01110100 and 11010101. A majority vote on each digit gives us the correct answer. This is known as an *error correcting code*. Of course, this method can fail. If we had received 01010001, 01110100 and 11010100 we would get the eighth digit wrong. We can increase our chances of getting the correct answer by increasing k, the number of repetitions.

There are better error correcting codes—they allow us to send shorter strings and still be at least as likely to be able to correct errors.

The basic idea is that we want to represent each of our characters by an n-string of zeroes and ones in such a way that if $a_1 a_2 \ldots a_n$ represents one character and $b_1 b_2 \ldots b_n$ represents another, then we often have $a_i \neq b_i$. Why is this good? It will help our discussion if we have some notation. Let A be the set of characters we are interested in and let f be a function that assigns to each $a \in A$ the n-string that will be used to represent a; i.e., $f(a) \in \{0,1\}^n$.

For $s, t \in \{0,1\}^n$, let $d(s,t)$ be the number of positions in which s differs from t. For example, if $a = 001001$ and $b = 000101$, then $d(a,b) = 2$. Finally, let $d(f)$ be the minimum of $d(f(x), f(y))$ over all $x \neq y$ in A. Whenever r and t differ in a position, either r and s differ in that position or s and t differ in that position. Thus

$$d(r,t) \ \leq \ d(r,s) + d(s,t). \tag{1.6}$$

We cannot replace the inequality with an equality, because r and t may agree in a position but both may differ from s in that position.

Suppose that $d(f) = 2$, that we transmit $f(x)$ and that a single zero-one bit is changed by static so that we receive s. We claim that we can tell an error has been made. If we can't tell, it must be because $f(y) = s$ for some $y \in A$. This is impossible because it would imply that $d(x,y) = 1$ and so $d(f) \leq 1$.

We can do more. Suppose that $d(f) = 3$, that we transmit $f(x)$ and that a single zero-one bit is changed by static so that we receive s. We claim that x is the only $y \in A$ such that $d(f(y), s) < 2$. In other words, x is the only character whose "encoding" is less than two errors away from s. Why is this? By (1.6) and the definition of $d(f)$, if $y \in A$ and $y \neq x$, then

$$3 \ = \ d(f) \ \leq \ d(f(x), f(y)) \ \leq \ d(f(x), s) + d(s, f(y)) \ = \ 1 + d(s, f(y)).$$

Thus $d(s, f(y)) \geq 2$.

More generally, if $d(f) \geq 2k+1$ and $s \in \{0,1\}^n$, there is at most one $x \in A$ with $d(f(x), s) \leq k$. Thus, if we assume that at most k errors have been made, we can recover the value of x. Given that s is received, one wants to find an $x \in A$ so that $d(f(x), s)$ is a minimum. This is called "decoding." Decoding efficiently is a difficult problem that we will not study.

Suppose we want $d(f) \geq 2k+1$, how large must n be? First we study lower bounds on n and then we study upper bounds.

Example 1.24 A lower bound on codeword length Here's the idea for finding a lower bound. Let $N(x)$ be the set of all n-strings s such that $d(f(x), s) \leq k$, where d and f are as defined in the preceding paragraphs. Later we will prove that $|N(x)|$ does not depend on x. Let $N = |N(x)|$. Suppose that $x \neq y \in A$. We will prove that $N(x) \cap N(y) = \emptyset$, the empty set. The number of n-strings must therefore be at least $N\,|A|$; however, there are 2^n n-strings and so $N\,|A| \leq 2^n$.

We now prove that $N(x) \cap N(y) = \emptyset$ using proof by contradiction. Suppose $s \in N(x) \cap N(y)$. Then $d(f(x), s) \leq k$ and $d(s, f(y)) \leq k$. By (1.6), $d(f(x), f(y)) \leq 2k$, contradicting $d(f) \geq 2k+1$.

We now compute $|N(x)|$ by noting that $s \in N(x)$ if and only if it differs from $f(x)$ in exactly j positions for some $j \leq k$. There are $\binom{n}{j}$ ways to select the j positions that must be changed. Thus

$$|N(x)| \ = \ \sum_{j=0}^{k} \binom{n}{j}.$$

Incidentally, this proves that $|N(x)|$ does not depend on x.

Dividing our inequality $N\,|A| \leq 2^n$ by N and substituting our formula for $N = |N(x)|$, we obtain

$$|A| \leq \frac{2^n}{\sum_{j=0}^{k} \binom{n}{j}}. \tag{1.7}$$

The smallest n for which (1.7) is true is a lower bound on how long the strings must be. Here are the lower bounds that are obtained for $k = 1$ and 2.

$\lvert A\rvert$	2	3	4	5	10	20
$k=1$	3	4	5	5	7	9
$k=2$	5	7	7	8	9	11

For example, if we have 20 characters and want to be able to correct strings that contain at most 2 errors, then the string length will have to be at least 11 (and possibly larger since this is only a lower bound).

The bound we obtained is called the "sphere packing bound" because $N(x)$ is thought of as a type of sphere with center x and radius k. □

We've shown that, if there is a code for A that corrects up to k errors, then the length n of the codewords must be so large that (1.7) holds. Now we want a result in the other direction; that is, we want an inequality such that, if n satisfies it, then there must be a code for A that corrects up to k errors. In other words, we want to find an upper bound on how large n must be. There are at least two ways to obtain such a result. One is to actually construct a code. Another is to show that among all possible codes for A having words of length n, at least one must be able to correct k and fewer errors. We'll take the second approach and use a probabilistic argument.

Example 1.25 An upper bound on codeword length We begin by constructing a probability space $(S, \Pr)$. Let S be all possible subsets of size $|A|$ of $\{0,1\}^n$. In other words, S consists of all $\binom{2^n}{|A|}$ possible sets of codewords. To make a subset into a code, we simply associate an element of the alphabet A with each element of the subset. Let Pr be the uniform distribution on S. Thus the elementary events are subsets which are potential codes; that is the $|A|$-subsets of $\{0,1\}^n$. A subset $C \in S$ will be *good* if every pair of its n-strings are at least distance $2k+1$ apart. (We've use C to remind us that the subset is a potential code.) Then assigning letters to n-strings in C will give us a code for A that corrects k and fewer errors. We want to find an upper bound on n. This will be an inequality on n such that S will contain at least one good subset if n satisfies the inequality.

Here is a method that is often used to obtain such inequalities. Let the random variable X be the number of pairs of bad n-strings in an randomly chosen C. Thus C is good whenever $X(C) = 0$. Since X must be a nonnegative integer, the expectation of X is

$$\mathbf{E}(X) \;=\; \sum_{k=0}^{\infty} k \,\Pr(X=k) \;\geq\; \Pr(X>0).$$

If we can prove that $\mathbf{E}(X) < 1$, we will have $\Pr(X>0) \;<\; 1$ and so $\Pr(X=0) \;>\; 0$. Since $X(C) = 0$ means C is good, there must be a good C. We'll evaluate $\mathbf{E}(X)$ in a minute, but first, what is the general method? Here it is.

- Introduce a probability space $(S, \Pr)$.
- Introduce a random variable X such that
  - the values of X are nonnegative integers,
  - if $X(s) = 0$, then $s \in S$ has the property we want.
- Find conditions such that $\mathbf{E}(X) < 1$.

We'll now do the last step, showing that $\mathbf{E}(X) < 1$.

Since our probability is uniform and since each $C \in S$ contains $\binom{|A|}{2}$ pairs of n-strings,

the expected value of X,
which is
the number of pairs of n-strings in a random C which are too close
equals
$\binom{|A|}{2}$ times the probability that two random n-strings are too close
which equals
$\binom{|A|}{2}$ times the probability that a new random n-strings is too close to a given one.

As in the preceding example, we want the number of n-strings within distance $2k$ of a given string, not counting the given string. You should be able to show that this equals $\sum_{i=1}^{2k} \binom{n}{i}$. Since there are

$2^n - 1$ strings other than the given one, we have

$$\mathbf{E}(X) \;=\; \binom{|A|}{2} \frac{\sum_{i=1}^{2k} \binom{n}{i}}{2^n - 1}.$$

We wanted this to be less than one. In other words, given $|A|$ and k, there will be a k-error correcting code with $|A|$ codewords if n is so large that

$$1 \;>\; \binom{|A|}{2} \frac{1}{2^n - 1} \sum_{i=1}^{2k} \binom{n}{i}. \tag{1.8}$$

Here is a table of the smallest values of n that satisfy the inequality for some values of $|A|$ and two values of k.

$\lvert A\rvert$	2	3	4	5	10	20
$k = 1$	3	7	8	9	12	15
$k = 2$	5	11	13	14	18	21

As you can see, these upper bounds are quite a bit larger than the lower bounds in the preceding example. □

One approach to creating a code would be to choose n so that the right side of (1.8) is not too close to 1. For example, say it equals 0.7. Since this number is $\mathbf{E}(X)$ which we saw was an upper bound on the probability that a random code is bad, there is a 30% probability that a randomly chosen element of S will be good. After a few random tries, we should be able to find a code. This seems to be an easy way to construct error correcting codes. It is—but they're no good! Why is this? With a random set of codewords, there is no problem encoding our message for transmission; however, if $|A|$ is large, it will be quite difficult to decode. To make decoding easy, one needs to construct a code that has some nice structure that one can use. This need has led to a considerable amount of research and to texts on the subject.

Exercises

1.3.1. Suppose that k and $n - k$ both get large as n gets large. Use Stirling's formula to show that

$$\binom{n}{k} \sim \frac{1}{\sqrt{2n\pi\lambda(1-\lambda)}} \left(\frac{1}{\lambda^\lambda (1-\lambda)^{1-\lambda}} \right)^n \quad \text{where } \lambda = k/n.$$

1.3.2. Suppose we have an election between two candidates and the ballots are counted one-by-one. At the end, the candidates are tied with n votes each. If the order of the votes is random, what is the probability that one of the candidates was never behind in the counting?
Hint. See Example 1.13.

1.3.3. How many 6 card hands contain 3 pairs?

1.3.4. How many ways can a 5 card hand containing 2 pairs be dealt? In other words, the *order* in which a person gets her cards matters.

1.3.5. How many 5 card hands contain a straight? A straight is 5 consecutive cards from the sequence A,2,3,4,5,6,7,8,9,10,J,Q,K,A without regard to suit.

1.3.6. How many compositions of n are there that have exactly k parts? The composition 1,2,2 of 5 has 3 parts.
Hint. See Exercise 1.1.4.

1.3.7. How many rearrangements of the letters in EXERCISES are there? How many arrangements of eight letters can be formed using the letters in EXERCISES? (No letter may be used more frequently than it appears in EXERCISES.)

1.3.8. In some card games only the values of the cards matter and their suits are irrelevant. Thus there are effectively only 13 distinct cards. How many different ways can a deck of cards be arranged in this case? The answer is a multinomial coefficient.

1.3.9. Return to choosing teams (Example 1.21). Suppose half the people are women and half are men, that each team must be as nearly evenly split as possible, and that there is one referee of each sex. How many ways can this be done?

1.3.10. There is an empire in the far away galaxy we've been visiting. They use the same alphabet (A,I,L,S,T) but their names consist of seven letters. Each name begins and ends with a consonant, contains no adjacent vowels and never contains three adjacent consonants. As before, if two consonants are adjacent, they cannot be the same.

(a) List the first 4 names in dictionary order.

(b) List the last 4 names in dictionary order.

(c) What are the first 4 names in dictionary order with just 2 vowels?

(d) How many names are possible?

*1.3.11. (*Multinomial Theorem*) Prove that the coefficient of $y_1^{m_1} y_2^{m_2} \cdots y_k^{m_k}$ in $(y_1 + y_2 + \cdots + y_k)^n$ is the multinomial coefficient $n!/m_1!m_2! \cdots m_k!$ when $n = m_1 + \cdots + m_k$ and zero otherwise.
Hint. Write

$$(y_1 + y_2 + \cdots + y_k)^n = \big((y_1 + y_2 + \cdots + y_{k-1}) + y_k\big)^n.$$

Now use the Binomial Theorem (Theorem 1.7) and induction on k.

1.3.12. Prove the following.

(a) $\displaystyle\binom{n}{k} = \binom{n}{n-k}$;

(b) $\displaystyle\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n} = 2^n$;

(c) $\displaystyle\binom{n}{0} - \binom{n}{1} + \cdots \pm \binom{n}{n} = 0 \quad \text{for} \quad n \geq 1$ (the signs alternate);

(d) $\displaystyle\binom{n+m}{k} = \binom{n}{0}\binom{m}{k} + \binom{n}{1}\binom{m}{k-1} + \cdots + \binom{n}{k}\binom{m}{0}$.

1.4 Recursions

Let's explore yet another approach to evaluating the binomial coefficient $C(n,k)$. As in the previous section, let $S = \{x_1, \ldots, x_n\}$. We'll think of $C(n,k)$ as counting k-subsets of S. Either the element x_n is in our subset or it is not. The cases where it is in the subset are all formed by taking the various $(k-1)$-subsets of $S - \{x_n\}$ and adding x_n to them. The cases where it is not in the subset are all formed by taking the various k-subsets of $S - \{x_n\}$. What we've done is describe how to build k-subsets of S from certain subsets of $S - \{x_n\}$. Since this gives each subset exactly once,

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

by the Rule of Sum.

The equation $C(n,k) = C(n-1,k-1) + C(n-1,k)$ is called a *recursion* because it tells how to compute $C(n,k)$ from values of the function with smaller arguments. This is a common approach which we can state in general form as follows.

Technique. Deriving recursions *Answering the question "How can I construct the things I want to count by using the same type of things of a smaller size?" usually gives a recursion.*

Sometimes it is easier to answer the question "How can I break the things I want to count up into smaller things of the same type?" This usually gives a recursion when it is turned around to answer the previous question.

Let's see how the second approach works for subsets. Given our collection of k-element subsets of S, throw out x_n if x_n is present. We obtain some $(k-1)$-element subsets of $S - \{x_n\}$ and some k-element subsets of $S - \{x_n\}$. In fact, you should be able to see that we obtain *all* $(k-1)$-element subsets and *all* k-element subsets exactly once. Turning this around gives us a way to build up k-element subsets of S.

We can use a recursion to compute a table of values by starting at the first row and computing new entries by adding previous ones. The arrows in Figure 1.3 show how this is done for the binomial coefficients. If the labels in this table are dropped, the rows are shifted slightly and a single 1 is added to the top row, then we obtain what is called *Pascal's triangle*. (See the figure.)

Actually, we've cheated a bit in all of this because the recursion only works when we have some values to start with. The correct statement of the recursion is either

$$\begin{aligned}
C(0,0) &= 1, \\
C(0,k) &= 0 \quad \text{for } k \neq 0 \quad \text{and} \\
C(n,k) &= C(n-1,k-1) + C(n-1,k) \quad \text{for } n > 0;
\end{aligned}$$

or

$$\begin{aligned}
C(1,0) &= C(1,1) = 1, \\
C(1,k) &= 0 \quad \text{for } k \neq 0,1 \quad \text{and} \\
C(n,k) &= C(n-1,k-1) + C(n-1,k) \quad \text{for } n > 1;
\end{aligned}$$

depending on whether we want to start with the row of Pascal's triangle consisting of 1 alone or the row consisting of 1,1. These starting values are called *initial conditions*. Note that, in either case, the last two conditions guarantee that $C(n,k) = 0$ for all $k < 0$.

		Values of k						
		0	1	2	3	4	5	6
	0	1	0	0	0	0	0	0
V	1	1	1	0	0	0	0	0
a		↓	↘	↓	↘	↓	↘	↓	↘	↓	↘	↓
l	2	1	2	1	0	0	0	0
u		↓	↘	↓	↘	↓	↘	↓	↘	↓	↘	↓
e	3	1	3	3	1	0	0	0
s		↓	↘	↓	↘	↓	↘	↓	↘	↓	↘	↓
	4	1	4	6	4	1	0	0
o		↓	↘	↓	↘	↓	↘	↓	↘	↓	↘	↓
f	5	1	5	10	10	5	1	0
		↓	↘	↓	↘	↓	↘	↓	↘	↓	↘	↓
n	6	1	6	15	20	15	6	1

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15   6   1
```

Figure 1.3 Left: The binomial coefficients are computed recursively. The columns of zeroes for $k < 0$ are omitted. Right: The results are arranged to give Pascal's triangle.

Example 1.26 Alternating subsets Let t_n be the number of subsets of $\{1, 2, \ldots, n\}$ such that, when the elements of the subset are listed in increasing order, the first is odd, the second is even, the third is odd, and so forth. We will allow the empty subset. Thus $t_0 = 1$ and $t_1 = 2$ because of $\emptyset$ and $\{1\}$. When $n = 4$ the subsets are

$$\emptyset \ \{1\} \ \{1,2\} \ \{1,2,3\} \ \{1,2,3,4\} \ \{1,4\} \ \{3\} \ \{3,4\},$$

and so $t_4 = 8$. Throwing out the subsets containing 4, we see that $t_3 = 5$. Throwing out those containing 3 or 4, we see that $t_2 = 3$.

How can we get a recursion for t_n? We can't simply take an acceptable subset for $n - 1$ and either add n to it or not. Why? For example, adding 4 to the subset $\{1, 2\}$ counted by t_3 would give $\{1, 2, 4\}$, which is not allowed. Of course, not adding an element is always safe. In other words, *every subset counted by t_n that does not contain n is counted by t_{n-1} and conversely.* If we can somehow figure a way to get the subsets counted by t_n that contain n, we'll be done.

Let's look again at the subsets for t_4 that contain 4. There are three of them:

$$\{1,2,3,4\}, \ \{1,4\} \text{ and } \{3,4\}.$$

Can we somehow reduce this to one or more groups of alternating subsets with $n < 4$? Since $t_2 = 3$, this might be a good place to start. To reduce our list to subsets counted by t_2, we'll need to throw out 3 and 4:

$$\{1,2\}, \ \{1\} \text{ and } \emptyset.$$

We've got the subsets counted by t_2. That's good, but can we reverse the process? Yes. Add 4 to each subset. If the resulting subset is not alternating, add 3 to it, too, and the result will be alternating. That's it!

Let's state it in general. We build the subsets counted by t_n in two ways.

(a) Take a subset counted by t_{n-1}.

(b) Take a subset S counted by t_{n-2}. Exactly one of the following will give a new alternating subset

  (i) Add n to S.

  (ii) Add $n - 1$ and n to S.

  In fact, if the largest element of S and n have the same parity (i.e., both odd or both even), we use (ii); if different parity, we use (i).

This requires $n \geq 2$ since we need to have $n-1 \geq 0$ and $n-2 \geq 0$ for (a) and (b) to make sense. You should be able to see that the procedure gives every alternating subset of $\{1, 2, \ldots, n\}$ exactly once. We've proved that

$$a_0 = 1, \quad a_1 = 2 \quad \text{and, for } n \geq 2, \quad a_n = a_{n-1} + a_{n-2}.$$

These are the *Fibonacci numbers*, which can be found in the index. □

Example 1.27 Set partitions A *partition of a set* B is a collection of nonempty subsets of B such that each element of B appears in exactly one subset. Each subset is called a *block* of the partition. The 15 partitions of $\{1, 2, 3, 4\}$ by number of blocks are

1 block:	$\{1,2,3,4\}$			
2 blocks:	$\{\{1,2,3\},\{4\}\}$	$\{\{1,2,4\},\{3\}\}$	$\{\{1,2\},\{3,4\}\}$	$\{\{1,3,4\},\{2\}\}$
	$\{\{1,3\},\{2,4\}\}$	$\{\{1,4\},\{2,3\}\}$	$\{\{1\},\{2,3,4\}\}$	
3 blocks:	$\{\{1,2\},\{3\},\{4\}\}$	$\{\{1,3\},\{2\},\{4\}\}$	$\{\{1,4\},\{2\},\{3\}\}$	$\{\{1\},\{2,3\},\{4\}\}$
	$\{\{1\},\{2,4\},\{3\}\}$	$\{\{1\},\{2\},\{3,4\}\}$		
4 blocks:	$\{\{1\},\{2\},\{3\},\{4\}\}$			

Let $S(n, k)$ be the number of partitions of an n-set having exactly k blocks. These are called *Stirling numbers of the second kind*.

> Do not confuse $S(n, k)$ with $C(n, k) = \binom{n}{k}$. In both cases we have an n-set. For $C(n, k)$ we want to *choose a subset* containing k elements and for $S(n, k)$ we want to *partition the set* into k blocks.

What is the value of $S(n, k)$? Let's try to get a recursion using the two questions in our technique.

How can we build partitions of $S = \{1, 2, \ldots, n\}$ with k blocks out of smaller cases? Using the approach we used for binomial coefficients, we'll take a partition of $S - \{n\}$ and add n to it somehow to get a k-block partition of S. If we take partitions of $\{1, 2, \ldots, n-1\}$ with $k-1$ blocks, we can simply add the block $\{n\}$. If we take partitions of $\{1, 2, \ldots, n-1\}$ with k blocks, we can add the element n to one of the k blocks. You should convince yourself that all k block partitions of $\{1, 2, \ldots, n\}$ arise in exactly one way when we do this. This gives us a recursion for $S(n, k)$. Putting n in a block by itself contributes $S(n-1, k-1)$. Putting n in a block with other elements contributes $S(n-1, k) \times k$ by the Rule of Product. By the Rule of Sum

$$S(n, k) \;=\; S(n-1, k-1) + k\, S(n-1, k). \tag{1.9}$$

We leave it to you to determine the values of n and k for which this is valid and to determine the initial conditions. You can construct the analog of Figure 1.3 as an exercise.

Now let's take the second question approach: How can we tear down a set partition into something smaller. As we did with subsets, we can simply remove n from our partition of $\{1, 2, \ldots, n\}$. You should convince yourself that this gives (1.9). There is another approach to tearing down: Instead of simply throwing out n, we can throw out the entire block containing n. If there are j elements in that block, throwing it out gives us a partition of an $(n-j)$-subset of $\{1, 2, \ldots, n-1\}$ into $k-1$ blocks. This gives all such partitions exactly once. Since there are $\binom{n-1}{n-j}$ ways to choose the subset, we have

$$S(n, k) \;=\; \sum_{j=1}^{n} \binom{n-1}{n-j} S(n-j, k-1) \quad \text{for } k > 1. \tag{1.10}$$

The initial conditions are $S(n, 1) = 1$ for $n \geq 1$ and $S(n, 1) = 0$ for $n \leq 0$.

At this point you may well expect us to come up with an explicit formula for $S(n, k)$ by a direct counting argument or a generating function argument since we did both for $C(n, k)$. These can both be done; however, more tools are required. They are developed in later chapters. Explicit formulas for $S(n, k)$ are not as nice as $C(n, k) = \frac{n!}{k!\,(n-k)!}$ since the simplest formula for $S(n, k)$ involves summation. □

So far all we've done is find recursions for various numbers and use the recursions to construct values. This is not the only way recursions can be used. Here are some others:

- Prove a formula, usually by induction: We'll see an example of this in a minute.
- Discover that two sets of numbers are the same because they have the same recursion. (Remember to include the initial conditions!)
- Study the numbers by looking directly at the recursion or by using generating functions: More on this in Part IV.

To illustrate a proof by induction, let's do Exercise 1.3.12(b), namely $\sum_{k=0}^{n}(-1)^k\binom{n}{k} = 2^n$ when $n \geq 0$. It's easy to check it for $n = 0$. Suppose $n > 0$ and the result is true for all values less than n. By the recursion

$$\sum_{k=0}^{n}\binom{n}{k} = \sum_{k=0}^{n}\left(\binom{n-1}{k-1}+\binom{n-1}{k}\right) = \sum_{k=0}^{n}\binom{n-1}{k-1}+\sum_{k=0}^{n}\binom{n-1}{k}.$$

Since the terms $\binom{n-1}{-1}$ and $\binom{n-1}{n}$ are zero, each of the last two sums is 2^{n-1} by the induction hypothesis and we are done since $2^{n-1}+2^{n-1}=2^n$.

Exercises

1.4.1. Calculate the next two rows in Pascal's Triangle.

1.4.2. Equation (1.9) gives a recursion for $S(n,k)$, but it is incomplete: initial conditions and the values of n and k for which it holds were omitted. Determine the values of n and k for which it is valid. Determine the initial conditions. Construct a table of values for $S(n,k)$ up through $n = 5$.

1.4.3. Derive a recursion like $S(n,k) = S(n-1,k-1)+kS(n-1,k)$ for ordered k-lists without repetitions that can be made from an n-set. Derive the recursion using an argument like that for $S(n,k)$; *do not* get the recursion using the formula $n!/(n-k)!$ that we found earlier. Since "like" is rather vague, there can be more than one solution to this exercise.

1.4.4. Exercise 1.3.12(c) you were asked to prove

$$\sum_{k=0}^{n}(-1)^k\binom{n}{k} = 0 \quad \text{for} \quad n \geq 1.$$

Prove it by induction on n using the recursion $\binom{n}{k} = \binom{n-1}{k-1}+\binom{n-1}{k}$.

1.4.5. For $n > 0$, prove the following formulas for $S(n,k)$.

$$S(n,n) = 1 \qquad S(n,n-1) = \binom{n}{2} \qquad S(n,1) = 1 \qquad S(n,2) = (2^n-2)/2$$

1.4.6. How can the initial conditions be set up to make (1.10) true for $n \geq 1$?

1.4.7. "Marking" something can help us derive a recursion. How many ways can we construct a k-subset of $\{1,2,\ldots,n\}$ and mark an element in the subset? You can do this in two ways:

- choose the subset and mark the element or
- choose the marked element and then choose the rest of the subset.

By counting these two ways, obtain the recursion $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$ for $k > 0$.

1.4.8. Let B_n be the total number of partitions of an n element set. Thus

$$B_n = S(n,0) + S(n,1) + \cdots + S(n,n).$$

(a) Prove that

$$B_{n+1} = \sum_{i=0}^{n} \binom{n}{i} B_{n-i},$$

where B_0 is defined to be 1.
Hint. Construct the block containing $n+1$ and then construct the rest of the partition.

(b) Calculate B_n for $n \leq 5$.

*1.4.9. Return to Exercise 1.2.13 (p. 18). You should have done (a) and (d) previously. Now you should be able to do (b) and (c) and obtain a recursion for (e). (Later, we will see how to use the "Principle of Inclusion and Exclusion" to obtain another solution for (e).)

1.4.10. We want to count the number of n digit sequences that have no adjacent zeroes. The digits must be chosen from the set $\{0, 1, \ldots, d-1\}$. For example, with $d = 3$ and $n = 4$, the sequences 0,2,1,0 and 2,1,2,2 are valid but 1,0,0,2 and 1,3,2,3 are not. Let the number of such sequences be A_n. (The case $d = 2$ is called the *Fibonacci numbers*.)

(a) From an n-sequence, remove the last digit if it is nonzero and the last two digits if the last digit is zero. By reversing this process, describe a way to build up all acceptable sequences by adding elements one or two at a time.

(b) Use (a) to obtain a recursion of the form $A_n = aA_{n-1} + bA_{n-2}$. What are a and b? For what n is the recursion valid? What are the initial conditions?

(c) Compute A_n for $n \leq 5$ when $d = 10$.

1.5 Multisets

Let $M(n,k)$ be the number of ways to choose k elements from an n-set when repetition is allowed and order doesn't matter. Will any of our three methods for handling $C(n,k)$ work for $M(n,k)$? Let's examine them.

- Imposing an order: The critical observation for our first method was that an unordered list can be ordered in $k!$ ways. This is not true if repetitions are allowed. To see this, note that the extreme case of k repetitions of one element has only one ordering.
- Using a recursion: We might be able to obtain a recursion, but we would still be faced with the problem of solving it.
- Using generating functions: To use the generating functions we have to allow for repetitions. This can be done very easily: Simply replace $(1 + x_i)$ in Example 1.14 (p. 19) with the infinite sum

$$1 + x_i + x_i^2 + x_i^3 + \cdots,$$

a geometric series which sums to $(1 - x_i)^{-1}$. Why does this replacement work? When we studied $C(n,k)$ in Example 1.14, the two terms in the factor $1 + x_i$ corresponded to not choosing the ith element OR choosing it, respectively. Now we need more terms: $x_i x_i$ for when the ith element is chosen to appear twice in our unordered list, $x_i x_i x_i$ for three appearances, and so forth. The distributive law still takes care of producing all possible combinations. As in Example 1.14, if we replace x_i by x for all i, the coefficient of x^k will be the number of multisets of size k. Thus $M(n,k)$ is the coefficient of x^k in $(1-x)^{-n}$. You should be able to use this fact and Taylor's Theorem to obtain $M(n,k) = (n+k-1)!/(n-1)!\,k!$. Thus

Theorem 1.8 Multiset formula *The number of k-multisets that can be made from an n-set is*

$$M(n,k) = \binom{n+k-1}{k}.$$

We can stop here since we have the answer; however, someone with an inquiring mind is likely not to. Such a person might ask "Why is $M(n,k)$ the number of ways to choose a k element subset of an $n-1+k$ element set?" Here "why" means an explanation that proves the two numbers are equal without actually counting. Posing and answering questions like this improve our understanding of a topic and improve our abilities to use the tools. We'll give one answer now. Another appears in the exercises.

Given a k-multiset of positive integers, list them in nondecreasing order, [2] say $a_1 \le a_2 \le \ldots \le a_k$. For each i, increase a_i by $i-1$ to obtain a new list. The new list consists of k distinct postive integers in increasing order. This sets up a one-to-one correspondence between multisets of positive integers and sets of positive integers.

What do the k-multisets formed from $\{1,2,\ldots,n\}$ correspond to? Since the largest element in the multiset is increased by $k-1$, each such multiset corresponds to a k-subset of $T = \{1,2,\ldots,n+k-1\}$. Conversely, every k-subset X of T corresponds to such a multiset: Simply list the elements of X in increasing order and subtract $i-1$ from the ith element for each i.

We have proved that in our one-to-one correspondence the multisets counted by $M(n,k)$ correspond to the sets counted by $C(n+k-1,k)$. Thus, these two numbers must be equal.

Example 1.28 Balls in boxes We are given 4 labeled boxes each of which can hold 2 balls and are also given 4 identical red balls and 4 identical green balls. How many ways can the balls be placed in the boxes?

This is not a problem that fits into our multiset model easily, although it can be made to fit. Nevertheless, it is the sort of problem that our methods work for. Indeed, it is very similar to the card hand problems. We'll look at it as if we hadn't seen those problems to emphasize the need to be able to translate problem descriptions without needing to force them into particular frameworks.

To begin with, we observe that once the red balls have been placed into boxes, there is only one way to place the green balls. (This is because there are exactly as many positions available as there are balls.) Thus, we can simply focus on placement of the red balls. Since there aren't very many ways to do that, we could simply list all of them. There is another approach that requires less work: First do the problem with unlabeled boxes and then label them. The unlabeled solutions are simply partitions of the number 4 into 4 parts, with zeroes allowed and no part exceeding 2. (A *partition of a number* is an unordered list that sums to the number.) The solutions are

$$1+1+1+1 \quad 0+1+1+2 \quad \text{and} \quad 0+0+2+2.$$

The first of these can be labeled in one way. The second can be labeled in 12 ways: choose the label for the empty box (4 ways) and then the label for the box containing 2 red balls (3 ways). The third solution can be labeled in $\binom{4}{2} = 6$ ways. Thus, there are $1+12+6 = 19$ solutions to the original problem. □

[2] A sequence is in nondecreasing order if the elements do not decrease as we move along the sequence. It is in increasing order if the elements increase. Thus $-7,3,5,6$ is both increasing and nondecreasing, $3,4,4,6$ is nondecreasing but not increasing, and $3,5,6,4$ is neither.

Given a set S, forming a k-subset or a k-multiset from S are two extremes: for a k-subset, no element can be repeated; for a k-multiset elements can be repeated as much as desired (as long as the total equals k). If we want something between the extremes, the counting is more difficult. For example, there's no simple formula for the number of k-multisets if each element appears at most j times except for $j = 1$ and $j \geq k$.

Exercises

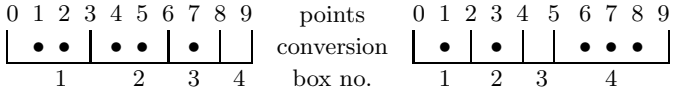
1.5.1. How many multisets can be formed from a set S if each element can appear at most j times? Your answer should be a simple formula.

1.5.2. It was stated in the preceding paragraph that "there's no simple formula for the number of k-multisets if each element appears at most j times except for $j = 1$ and $j \geq k$." What are the formulas for $j = 1$ and $j \geq k$?

1.5.3. Without using the formula for $M(n, k)$, prove that $M(n, k) = M(n-1, k) + M(n, k-1)$. What are the initial conditions for this recursion?

1.5.4. Prove that $M(n, k)$ is the number of ways to place k indistinguishable balls into n boxes.
Hint. If you have $n = 7$ boxes and $k = 8$ balls, the list 1,1,1,2,4,4,4,7 can be interpreted as "Place three balls in box 1, one ball in box 2, three balls in box 4 and one ball in box 7."

1.5.5. Imagine $\{1, 2, ..., n + k - 1\}$ represented as points on a line in the usual way. Convert $n - 1$ of the points to vertical bars and convert 0 and $n + k$ to vertical bars. Combine this with the previous problem to prove that $M(n, k) = C(n+k-1, n-1)$. This gives one answer the the question of "why" the two numbers are equal.
Hint. Here are examples of a correspondence with 5 balls and 4 boxes



1.5.6. Prove that the number of unordered k-lists made from n different items and using each item at most twice is the coefficient of x^k in $(1 + x + x^2)^n$. Generalize this.

1.5.7. Let $T(n, k)$ be the the number of k-multisets made from n different items, using each item at most twice in a multiset. Prove that

$$T(n, k) = T(n-1, k) + T(n-1, k-1) + T(n-1, k-2).$$

Relate this problem to the previous exercise and generalize it.

1.5.8. Prove by induction on n and k that the number of k-multisets that can be formed from an n-set is $\binom{n+k-1}{k}$. Let the answer be $M(n, k)$. To start the induction, verify the formula for $M(1, k)$ and for $M(n, 1)$ for all n and k. For the induction step, use $M(n, k-1)$ and $M(n-1, k)$ to derive $M(n, k)$.

1.5.9. Let $f(b, t)$ be the number of ways to put b labeled balls into t labeled tubes. When balls are put into tubes the order matters: because the diameter of the tube is only slightly larger than that of the balls, the balls end up stacked on top of each other in a tube.

(a) Prove by induction on b that $f(b, t) = t(t+1)\cdots(t+b-1)$. (To do this, you will first need a recursion for $f(b, t)$.)
Hint. There are at least two ways to get a recursion on b: (i) insert $b-1$ balls and then the last or (ii) insert the first ball and then the remaining $b-1$.

(b) Give a noninductive combinatorial proof of the formula for $f(b, t)$.

*1.5.10. Let $f(n,k)$ be the number of ways to partition an n-set into k nonempty blocks where the order of the entries in a block matters but the order of the blocks does not matter.

(a) Prove by induction that

$$f(n,k) = \frac{n!}{k!}\binom{n-1}{k-1}.$$

Hint. An argument like the one leading to (1.9) can be used for the induction step.

(b) Give a noninductive combinatorial proof of the formula for $f(n,k)$.

Notes and References

We conclude this chapter with a table of the numbers of each of the four basic types of lists; i.e., ordered and unordered with repetitions allowed or forbidden. It is given in Figure 1.4. We are selecting k things from an n-set. The rules governing the selections are listed at the top and the left of the figure. As indicated in the figure, these numbers can also be interpreted in terms of placing balls into labeled boxes.

The concepts in the Rules of Sum and Product were known in ancient times. Until fairly recently, combinatorics has been synonymous with counting. This may be due to its connections with probability theory. You can learn about this in many books, but it is hard to do better than Feller's classic text [7]. We will focus on enumeration problems again in Chapters 4, 10 and 11.

Enumeration is still an active area of research in combinatorics. Although much of the research uses more sophisticated tools (See the notes to Chapters 4, 10 and 11.), some current research relies only on clever elementary arguments. You may be able to find some papers of this sort by browsing through such combinatorial journals as the *Journal of Combinatorial Theory, Series A*, the *European Journal of Combinatorics*, and *The Electronic Journal of Combinatorics* (at `http://www.combinatorics.org`). Unfortunately, proofs are often given rather tersely and careless authors sometimes neglect to explain terminology. As examples of short papers that you may be able to read now, you may want to look at [6] and [10]. Lubell's proof, which was used in Example 1.22, appeared in [9].

Because of the fundamental importance of counting, it is discussed in almost every text whose title refers to combinatorics or discrete mathematics. A few of the texts with material around the level of this book are those by Biggs [2; Ch.3], Bogart [3; Chs.1, 2], Cohen [4; Chs.2, 4], Stanton and White [12; Ch.1] and Tucker [13; Ch.5]. More advanced treatments can be found in the books by Comtet [5], Goulden and Jackson [8] and Stanley [11]. Anderson [1] starts off with Lubell's proof of Sperner's Theorem (Example 1.22) and then continues with other topics related to subsets of sets. His text is an example of the breadth of combinatorics—it does not discuss enumeration and has practically no overlap with our text.

Many papers have been written on Catalan numbers. Stanley [11, v.2] lists sixty-six things counted by Catalan numbers in his Exercise 6.1.9 (pp.219–229) and gives a partial solution to the exercise (pp.256–265).

Derivations of Stirling's formula (Theorem 1.5 (p. 12)) can be found in many places, including Feller's text [7; II.9, VII.2].

1. Ian Anderson, *Combinatorics of Finite Sets*, Dover (2002).
2. Norman L. Biggs, *Discrete Mathematics*, 2nd ed., Oxford Univ. Press (2003).
3. Kenneth P. Bogart, *Introductory Combinatorics*, 3rd ed., Brooks/Cole (2000).
4. Daniel I.A. Cohen *Basic Techniques of Combinatorial Theory*, John Wiley (1978).
5. Louis Comtet, *Advanced Combinatorics: The Art of Finite and Infinite Expansions*, Reidel (1974).

	Ordered (labeled balls)	Unordered (unlabeled balls)
Repetitions Allowed	Lists with repetition n^k	Multisets $\frac{(n+k-1)!}{k!(n-1)!}$
Repetitions Forbidden $\left(\begin{array}{c}\text{at most one}\\ \text{ball per box}\end{array}\right)$	Lists of distinct elements $\frac{n!}{(n-k)!}$	Sets $\frac{n!}{k!(n-k)!}$

Figure 1.4 The four basic list enumerators for k-lists made from n-sets. They can also be interpreted as placing k balls (either labeled or unlabeled) into n labeled boxes. The ball and box interpretation is indicated parenthetically.

6. Paul Erdős and Joel Spencer, Monochromatic sumsets, *J. Combinatorial Theory, Series A* **50** (1989), 162–163.
7. William Feller, *An Introduction to Probability Theory and Its Applications*, 3rd ed., John Wiley (1968).
8. Ian P. Goulden and David M. Jackson, *Combinatorial Enumeration*, Dover (2004). Reprint of John Wiley edition (1983).
9. David Lubell, A short proof of Sperner's Lemma, *J. Combinatorial Theory* **1** (1966), 299.
10. Albert Nijenhuis and Herbert S. Wilf, A method and two algorithms on the theory of partitions, *J. Combinatorial Theory, Series A* **18** (1975), 219–222.
11. Richard P. Stanley, *Enumerative Combinatorics*, vols. 1 and 2, Cambridge Univ. Press (1999, 2001).
12. Dennis Stanton and Dennis White, *Constructive Combinatorics*, Springer-Verlag (1986).
13. Alan C. Tucker, *Applied Combinatorics*, 3rd ed., John Wiley (2001).

CHAPTER 2

Functions

Introduction

Functions play a fundamental role in nearly all of mathematics. Combinatorics is no exception. In the next section we review the basic terminology and notation for functions. Permutations are special functions that arise in a variety of ways in combinatorics. Besides studying them for their own interest, we'll see them as a central tool in counting objects with symmetries and as a source of examples for decision trees in later chapters. Section 3 discusses some combinatorial aspects of functions that relate to some of the material in the previous chapter. We conclude the chapter with a discussion of Boolean functions. They form the mathematical basis of most computer logic.

2.1 Some Basic Terminology

Terminology for Sets

Except for the real numbers ($\mathbb{R}$), rational numbers ($\mathbb{Q}$) and integers ($\mathbb{Z}$), our sets are normally finite. The set of the first n positive integers, $\{1, 2, \ldots, n\}$ will be denoted by $\underline{n}$.

Recall that $|A|$ is the number of elements in the set A. When it is convenient to do so, we'll assume that the elements of a set A have been linearly ordered and denote the ordering by $a_1, a_2, \ldots, a_{|A|}$. Unless clearly stated otherwise, the ordering on a set of numbers is the numerical ordering. For example, the ordering on $\underline{n}$ is $1, 2, 3, \ldots, n$.

If A and B are sets, we write $A - B$ for the set of elements in A that are not in B:

$$A - B \;=\; \{\, x \mid x \in A \text{ and } x \notin B \,\}.$$

(This is also written $A \backslash B$.)

If A and B are sets, recall from the previous chapter that the Cartesian product $A \times B$ is the set of all ordered pairs built from A and B:

$$A \times B \;=\; \{\, (a, b) \mid a \in A \text{ and } \; b \in B \,\}.$$

We also call $A \times B$ the *direct product* of A and B.

If $A = B = \mathbb{R}$, the real numbers, then $\mathbb{R} \times \mathbb{R}$, written $\mathbb{R}^2$ is frequently interpreted as coordinates of points in the plane. Two points are the same if and only if they have the same coordinates, which says the same thing as our definition of $(a, b) = (a', b')$. Recall that the direct product can be extended to any number of sets. How can $\mathbb{R} \times \mathbb{R} \times \mathbb{R} = \mathbb{R}^3$ be interpreted?

What are Functions?

Definition 2.1 Function *If A and B are sets, a* **function** *from A to B is a rule that tells us how to find a unique $b \in B$ for each $a \in A$. We write $f\colon A \to B$ to indicate that f is a function from A to B. We call the set A the* **domain** *of f and the set B the* **codomain** *of f. To specify a function completely you must give its domain, codomain and rule. The set of all functions from A to B is written B^A, for a reason we will soon explain. Thus $f\colon A \to B$ and $f \in B^A$ say the same thing.*

In calculus you dealt with functions whose codomains were $\mathbb{R}$ and whose domains were contained in $\mathbb{R}$; for example, $f(x) = 1/(x^2-1)$ is a function from $\mathbb{R} - \{-1, 1\}$ to $\mathbb{R}$. You also studied functions of functions! The derivative is a function whose domain is all differentiable functions and whose codomain is all functions. If we wanted to use functional notation we could write $D(f)$ to indicate the function that the derivative associates with f. Can you see how to think of the integral as a function? This is a bit tricky because of the constant of integration. We won't pursue it.

Definition 2.2 One-line notation *When A is ordered, a function can be written in* **one-line notation** *as $(f(a_1), f(a_2), \ldots, f(a_{|A|}))$. Thus we can think of function as an element of $B \times B \times \ldots \times B$, where there are $|A|$ copies of B. Instead of writing $B^{|A|}$ to indicate the set of all functions, we write B^A. Writing $B^{|A|}$ is incomplete because the domain is not specified. Instead, only its size is given.*

Example 2.1 Using the notation To get a feeling for the notation used to specify a function, it may be helpful to imagine that you have an envelope or box that contains a function. In other words, this envelope contains all the information needed to completely describe the function. Think about what you're going to see when you open the envelope.

* * * Stop and think about this! * * *

You might see

$$P = \{a, b, c\}, \qquad g\colon P \to \underline{4}, \qquad g(a) = 3, \quad g(b) = 1 \quad \text{and} \quad g(c) = 4.$$

This tells you that the name of the function is g, the domain of g is P, which is $\{a, b, c\}$, and the codomain of g is $\underline{4} = \{1, 2, 3, 4\}$. It also tells you the values in $\underline{4}$ that g assigns to each of the values in its domain. Someone else may have put

$$g\colon \{a, b, c\} \to \underline{4}, \qquad \text{ordering: } a, b, c, \qquad g = (3, 1, 4).$$

in the envelope instead. This describes the same function but doesn't give a name for the domain. On the other hand, it gives an order on the domain so that the function can be given in one line form. Can you describe other possible envelopes for the same function?

What if the envelope contained only $g = (3, 1, 4)$? You've been cheated! You *must* know the domain of g in order to known what g is. What if the envelope contained

$$\text{the domain of } g \text{ is } \{a, b, c\}, \qquad \text{ordering: } a, b, c, \qquad g = (3, 1, 4)?$$

We haven't specified the codomain of g, but is it necessary since we know the values of the function? Our definition included the requirement that the codomain be specified, so this is not a complete definition. On the other hand, we frequently only need to know which values in its codomain g actually takes on (here 1, 3 and 4), so we'll be sloppy in such cases and accept this as if it were a complete specification. □

Example 2.2 Counting functions By the Rule of Product, $\left|B^A\right| = |B|^{|A|}$. We can represent a subset S of A by a unique function $f: A \to \underline{2}$ where $f(x) = 1$ if $a \notin S$ and $f(x) = 2$ if $x \in S$. This proves that there are $2^{|S|}$ such subsets. We can represent a list of k elements of a set S with repetition allowed by a unique function $f: \underline{k} \to S$. In this representation, the list corresponds to the function written in one line notation. (Recall that the ordering on $\underline{k}$ is the numerical ordering.) This proves that there are exactly $|S|^k$ such lists. □

Definition 2.3 Types of functions *Let $f: A \to B$ be a function. If for every $b \in B$ there is an $a \in A$ such that $f(a) = b$, then f is called a* **surjection** *(or an* **onto function***). Another way to describe a surjection is to say that it takes on each value in its codomain at least once.*

If $f(x) = f(y)$ implies $x = y$, then f is called an **injection** *(or a* **one-to-one function***). Another way to describe an injection is to say that it takes on each value in its codomain at most once. The injections in $S^{\underline{k}}$ correspond to lists without repetitions.*

If f is both an injection and a surjection, it is a called a **bijection***. The bijections of A^A are called the* **permutations** *of A. If $f: A \to B$ is a bijection, we may talk about the* **inverse** *of f, written f^{-1}, which reverses what f does. Thus $f^{-1}: B \to A$ and $f^{-1}(b)$ is that unique $a \in A$ such that $f(a) = b$. Note that $f(f^{-1}(b)) = b$ and $f^{-1}(f(a)) = a$. Do not confuse f^{-1} with $1/f$. For example, if $f: \mathbb{R} \to \mathbb{R}$ is given by $f(x) = x^3+1$, then $1/f(x) = 1/(x^3+1)$ and $f^{-1}(y) = (y-1)^{1/3}$.*

Example 2.3 Using the notation We'll illustrate the ideas in the previous paragraph. Let $A = \underline{4}$, $B = \{a, b, c, d, e\}$ and $f = (d, c, d, a)$. Since the value d is taken on twice by f, f is not an injection. Since the value b is not taken on by f, f is not a surjection. (We could have said e is not taken on, instead.) The function (b, d, c, e) is an injection since there are no repeats in the list of values taken on by the function.

Now let $A = \underline{4}$, $B = \{x, y, z\}$ and $g = (x, y, x, z)$. Since every element of B appears at least once in the list of values taken on, f is a surjection.

Finally, let $A = B = \underline{4}$ and $h = (3, 1, 4, 2)$. The function is both an injection and a surjection. Hence, it is a bijection. Since the domain and codomain are the same and f is a bijection, it is a permutation of $\underline{4}$. The inverse of h is $(2, 4, 1, 3)$. □

Example 2.4 Two-line notation Since one line notation is a simple, brief way to specify functions, we'll use it frequently. If the domain is not a set of numbers, the notation is poor because we must first pause and order the domain. There are other ways to write functions which overcome this problem. For example, we could write $f(a) = 4$, $f(b) = 3$, $f(c) = 4$ and $f(d) = 1$. This could be shortened up somewhat to $a \to 4$, $b \to 3$, $c \to 4$ and $d \to 1$. By turning each of these sideways, we can shorten it even more: $\left(\begin{smallmatrix} a & b & c & d \\ 4 & 3 & 4 & 1 \end{smallmatrix}\right)$ For obvious reasons, this is called *two-line notation*. Since x always appears directly over $f(x)$, there is no need to order the domain; in fact, we need not even specify the domain separately since it is given by the top line. If the function is a bijection, its inverse is obtained by interchanging the top and bottom lines.

The arrows we introduced in the last paragraph can be used to help visualize different properties of functions. Imagine that you've listed the elements of the domain A in one column and the elements of the codomain B in another column to the right of the the domain. Draw an arrow from a to b if $f(a) = b$. Thus the heads of arrows are labeled with elements of B and the tails with elements of A. Since f is a function, no two arrows have the same tail. If f is an injection, no two arrows have the same head. If f is a surjection, every element of B is on the head of some arrow. You should be able to describe the situation when f is a bijection. □

Exercises

2.1.1. This exercise lets you check your understanding of the definitions. In each case below, some information about a function is given to you. Answer the following questions and give reasons for your answers: (The answers are given at the end of this problem set.)

(i) Have you been given enough information to specify the function; i.e., would this be enough data for a function envelope?

(ii) Can you tell whether or not the function is an injection? a surjection? a bijection? If so, what is it?

(iii) If possible, give the function in two line form.

(a) $f \in \underline{3}^{\{\heartsuit,\clubsuit,\diamondsuit,\spadesuit\}}$, $\quad f = (3, 1, 2, 3)$.

(b) $f \in \{\heartsuit, \clubsuit, \diamondsuit, \spadesuit\}^{\underline{3}}$, $\quad f = (\spadesuit, \heartsuit, \clubsuit)$.

(c) $f \in \underline{4}^{\underline{3}}$, $\quad 2 \to 3, \quad 1 \to 4, \quad 3 \to 2$.

2.1.2. Let A and B be finite sets and $f\colon A \to B$. Prove the following claims. Some are practically restatements the of definitions, some require a few steps.

(a) If f is an injection, then $|A| \leq |B|$.

(b) If f is a surjection, then $|A| \geq |B|$.

(c) If f is a bijection, then $|A| = |B|$.

(d) If $|A| = |B|$, then f is an injection if and only if it is a surjection.

(e) If $|A| = |B|$, then f is a bijection if and only if it is an injection or it is a surjection.

Answers

2.1.1. (a) We know the domain and codomain of f. By Exercise 2, f cannot be an injection. Since no order is given for the domain, the attempt to specify f in one-line notation is meaningless. If the attempt at specification makes any sense, it tells us that f is a surjection. We cannot give it in two line form since we don't know the function.

(b) We know the domain and codomain of f and the domain has an implicit order. Thus the one-line notation specifies f. It is an injection but not a surjection. In two line form it is $\left(\begin{smallmatrix} 1 & 2 & 3 \\ \spadesuit & \heartsuit & \clubsuit \end{smallmatrix}\right)$.

(c) This function is specified and is an injection. In one-line notation it would be (4,3,2), and, in two line notation, $\left(\begin{smallmatrix} 1 & 2 & 3 \\ 4 & 3 & 2 \end{smallmatrix}\right)$.

2.2 Permutations

Before beginning our discussion, we need the notion of composition of functions. Suppose that f and g are two functions such that the values f takes on are contained in the domain of g. We can write this as $f\colon A \to B$ and $g\colon C \to D$ where $f(a) \in C$ for all $a \in A$. We define the *composition* of g and f, written $gf\colon A \to D$ by $(gf)(x) = g(f(x))$ for all $x \in A$. The notation $g \circ f$ is also used to denote composition. Suppose that f and g are given in two line notation by

$$f = \begin{pmatrix} p & q & r & s \\ P & R & T & U \end{pmatrix} \qquad g = \begin{pmatrix} P & Q & R & S & T & U & V \\ 1 & 3 & 5 & 2 & 4 & 6 & 7 \end{pmatrix}.$$

Then $gf = \begin{pmatrix} p & q & r & s \\ 1 & 5 & 4 & 6 \end{pmatrix}$.

The set of permutations on a set A is denoted in various ways in the literature. Two notations are $\mathrm{PER}(A)$ and $\mathcal{S}(A)$. Suppose that f and g are permutations of a set A. Recall that a permutation is a bijection from a set to itself and so it makes sense to talk about f^{-1} and fg. We claim that fg and f^{-1} are also permutations of A. This is easy to see if you write the permutations in two-line form and note that the second line is a rearrangement of the first if and only if the function is a permutation.

Again suppose that f is a permutation. Instead of $f \circ f$ or ff we write f^2. Note that $f^2(x)$ is not $(f(x))^2$. (In fact, if multiplication is not defined in A, $(f(x))^2$ has no meaning.) We could compose three copies of f. The result is written f^3. In general, we can compose k copies of f to obtain f^k. A cautious reader may be concerned that $f \circ (f \circ f)$ may not be the same as $(f \circ f) \circ f$. They are equal. In fact, $f^{k+m} = f^k \circ f^m$ for all nonnegative integers k and m, where f^0 is defined by $f^0(x) = x$ for all x in the domain. This is based on the "associative law" which states that $f \circ (g \circ h) = (f \circ g) \circ h$ whenever the compositions make sense. We'll prove these results.

To prove that the two functions are equal, it suffices to prove that they take on the same values for all x in the domain. Let's use this idea for $f \circ (g \circ h)$ and $(f \circ g) \circ h$. We have

$$\begin{aligned} (f \circ (g \circ h))(x) &= f((g \circ h)(x)) && \text{by the definition of } \circ, \\ &= f(g(h(x))) && \text{by the definition of } \circ. \end{aligned}$$

Similarly

$$\begin{aligned} ((f \circ g) \circ h)(x) &= (f \circ g)(h(x)) && \text{by the definition of } \circ, \\ &= f(g(h(x))) && \text{by the definition of } \circ. \end{aligned}$$

More generally, one can use this approach to prove by induction that $f_1 \circ f_2 \circ \cdots \circ f_n$ is well defined. This result then implies that $f^{k+m} = f^k \circ f^m$. Note that we have proved that the associative law for any three functions f, g and h for which the domain of f contains the values taken on by g and the domain of g contains the values taken on by h.

Example 2.5 Using two-line and composition notations Let f and g be the permutations

$$f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 5 & 3 \end{pmatrix} \qquad g = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \end{pmatrix}.$$

We can compute fg by calculating all the values. This can be done fairly easily from the two line form: For example, $(fg)(1)$ can be found by noting that the image of 1 under g is 2 and the image of 2 under f is 1. Thus $(fg)(1) = 1$. You should be able to verify that

$$fg = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 5 & 3 & 2 \end{pmatrix} \quad \text{and} \quad gf = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 2 & 5 & 1 & 4 \end{pmatrix} \neq fg$$

and that

$$f^2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 5 & 3 & 4 \end{pmatrix} \quad f^3 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 3 & 4 & 5 \end{pmatrix} \quad \text{and} \quad g^5 = f^6 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}.$$

Note that it is easy to get the inverse, simply interchange the two lines. Thus

$$f^{-1} = \begin{pmatrix} 2 & 1 & 4 & 5 & 3 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} \quad \text{which is the same as } f^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 5 & 3 & 4 \end{pmatrix},$$

since the order of the columns in two line form does not matter. □

Let f be a permutation of the set A and let $n = |A|$. If $x \in A$, we can look at the sequence $x, f(x), f(f(x)), \ldots, f^k(x), \ldots$, which is often written as $x \to f(x) \to f(f(x)) \to \cdots \to f^k(x) \to \cdots$. Since the codomain of f has n elements, this sequence will contain a repeated element in the first $n+1$ entries. Suppose that $f^s(x)$ is the first sequence entry that is ever repeated and that $f^{s+p}(x)$ is the first time that it is repeated. Apply $(f^{-1})^s$ to both sides of this equality to obtain $x = f^p(x)$ and so, in fact, $s = 0$. It follows that the sequence cycles through a pattern of length p forever since $f^{p+1}(x) = f(f^p(x)) = f(x)$, $f^{p+2}(x) = f^2(f^p(x)) = f^2(x)$, and so on. We call $(x, f(x), \ldots, f^{p-1}(x))$ the *cycle* containing x and call p the *length of the cycle*. If a cycle has length p, we call it a p-cycle. Cyclic shifts of a cycle are considered the same; for example, if (1,2,6,3) is the cycle containing 1 (as well as 2, 3 and 6), then (2,6,3,1), (6,3,1,2) and (3,1,2,6) are other ways of writing the cycle.

Example 2.6 Using cycle notation Consider the permutation $f = \left(\begin{smallmatrix} 1&2&3&4&5&6&7&8&9 \\ 2&4&8&1&5&9&3&7&6 \end{smallmatrix}\right)$. Since $1 \to 2 \to 4 \to 1$, the cycle containing 1 is (1,2,4). We could equally well write it (2,4,1) or (4,1,2); however, (1,4,2) is different since it corresponds to $1 \to 4 \to 2 \to 1$ The usual convention is to list the cycle starting with its smallest element. The cycles of f are (1,2,4), (3,8,7), (5) and (6,9). We write f in *cycle form* as

$$f = (1,2,4)\ (3,8,7)\ (5)\ (6,9).$$

It is common practice to omit the cycles of length one and write $f = (1,2,4)(3,8,7)(6,9)$. The inverse of f is obtained by reading the cycles backwards because $f^{-1}(x)$ is the lefthand neighbor of x in a cycle. Thus

$$f^{-1} = (4,2,1)(7,8,3)(9,6) = (1,4,2)(3,7,8)(6,9). \quad \square$$

Cycle form is useful in certain aspects of the branch of mathematics called "finite group theory." We will find it useful later when we study the problem of counting structures having symmetries. Here's an application now.

Example 2.7 Powers of permutations With a permutation in cycle form, its very easy to calculate a power of the permutation. For example, suppose we want the tenth power of the permutation whose cycle form (including cycles of length 1) is $(1,5,3)(7)(2,6)$. To find the image of 1, we take ten steps: $1 \to 5 \to 3 \to 1 \cdots$. Where does it stop after ten steps? Since three steps bring us back to where we started (because 1 is in a cycle of length three), nine steps take us around the cycle three times and the tenth takes us to 5. Thus $1 \to 5$ in the tenth power. Similarly, $5 \to 3$ and $3 \to 1$. Clearly $7 \to 7$ regardless of the power. Ten steps take us around the cycle $(2,6)$ exactly five times, so $2 \to 2$ and $6 \to 6$. Thus the tenth power is $(1,5,3)(7)(2)(6)$.

Suppose we have a permutation in cycle form whose cycle lengths all divide k. The reasoning in the previous paragraph shows that the kth power of that permutation will be the identity; that is, all the cycles will be 1-long and so every element is mapped to itself. In particular, if we are considering permutations of an n-set, every cycle has length at most n and so we can take $k = n!$ regardless of the permutation. We have shown

Theorem 2.1 *Given a set S, there is a k depending on $|S|$ such that f^k is the identity map for every permutation f of S.*

Without cycle notation, it would be harder to prove the theorem. $\square$

Example 2.8 Average cycle information What is the average number of elements fixed by a random permutation? More generally, what is the average number that belong to cycles of length k? What is the average number of cycles in a random permutation? We'll see that these questions are easy to answer.

Make the $n!$ permutations of $\{1, 2, \ldots, n\}$ into a probability space by giving each permutation probability $1/n!$—the uniform distribution. We want to define random variables X_i, one for each element of $\underline{n}$ such that their sum is the number of elements in k-cycles. Then $\mathbf{E}(X_1 + \cdots + X_n)$ is the average number of elements that belong to k-cycles. Let

$$X_i \;=\; \begin{cases} 1, & \text{if } i \text{ belongs to a } k\text{-cycle,} \\ 0, & \text{otherwise.} \end{cases}$$

Note that $\mathbf{E}(X_i)$ is the same for all i. By properties of expectation

$$\mathbf{E}(X_1 + \cdots + X_n) \;=\; \mathbf{E}(X_1) + \cdots + \mathbf{E}(X_n) \;=\; n\,\mathbf{E}(X_1).$$

It is shown in Exercise 2.2.2 that there are $(n-1)!$ permutations with $X_1 = 1$. Since each permutation has probability $1/n!$, $n\,\mathbf{E}(X_1) = n\frac{(n-1)!}{n!} = 1$. In other words, on average one element belongs to a k-cycle, regardless of the value of k as long as $1 \le k \le n$.

We now consider the average number of cycles in a permutation. To do this we want to define random variables, one for each element of $\underline{n}$, such that their sum is the number of cycles. Let

$$Y_i \;=\; \frac{1}{\text{length of the cycle containing } i}.$$

Since each element of a k-cycle contributes $1/k$ to the sum $Y_1 + \cdots + Y_n$, all k elements in the cycle contribute a total of 1. Thus $Y_1 + \cdots + Y_n$ is the number of cycles in the permutation. We have

$$\begin{aligned} \mathbf{E}(Y_1 + \cdots + Y_n) \;&=\; \mathbf{E}(Y_1) + \cdots + \mathbf{E}(Y_n) \;=\; n\,\mathbf{E}(Y_1) \\ &=\; n \sum_{k=1}^{n} \Pr(1 \text{ is in a } k\text{-cycle})\,\frac{1}{k} && \text{by definition of } \mathbf{E} \\ &=\; n \sum_{k=1}^{n} \frac{1}{n}\,\frac{1}{k} && \text{by Exercise 2.2.2} \\ &=\; \sum_{k=1}^{n} \frac{1}{k}. \end{aligned}$$

The last sum is approximately $\ln n$ because the sum is approximately $1 + \int_1^n \frac{dx}{x} \;=\; 1 + \ln n$. □

***Example 2.9 Involutions** An *involution* is a permutation which is equal to its inverse. Since $f(x) = f^{-1}(x)$, we have $f^2(x) = f(f^{-1}(x)) = x$. Thus involutions are those permutations which have all their cycles of lengths one and two. How many involutions are there on $\underline{n}$?

Let's count the involutions with exactly k 2-cycles and use the Rule of Sum to add up the results. We can build such an involution as follows:

- Select $2k$ elements for the 2-cycles AND
- partition these $2k$ elements into k blocks that are all of size 2 AND
- put the remaining $n - 2k$ elements into 1-cycles.

Since there is just one 2-cycle on two given elements, we can interpret each block as 2-cycle. This specifies f. The number of ways to carry out the first step is $\binom{n}{2k}$. The next step is trickier. A first guess might be simply the multinomial coefficient $\binom{2k}{2,\ldots,2} = (2k)!/2^k$. This leads to the dilemma of the poker hand with two pairs (Example 1.16): We're assuming an ordering on the pairs even though

they don't have one. For example, with $k = 3$ and the set $\underline{6}$, there are just 15 possible partitions as follows.

$$\begin{array}{lll}
\{\{1,2\},\{3,4\},\{5,6\}\} & \{\{1,2\},\{3,5\},\{4,6\}\} & \{\{1,2\},\{3,6\},\{4,5\}\} \\
\{\{1,3\},\{2,4\},\{5,6\}\} & \{\{1,3\},\{2,5\},\{4,6\}\} & \{\{1,3\},\{2,6\},\{4,5\}\} \\
\{\{1,4\},\{2,3\},\{5,6\}\} & \{\{1,4\},\{2,5\},\{3,6\}\} & \{\{1,4\},\{2,6\},\{3,5\}\} \\
\{\{1,5\},\{2,3\},\{4,6\}\} & \{\{1,5\},\{2,4\},\{3,6\}\} & \{\{1,5\},\{2,6\},\{3,4\}\} \\
\{\{1,6\},\{2,3\},\{4,5\}\} & \{\{1,6\},\{2,4\},\{3,5\}\} & \{\{1,6\},\{2,5\},\{3,4\}\}
\end{array}$$

This is smaller than $\binom{6}{2,2,2} = 6!/2!2!2! = 90$ because all 3! ways to order the three blocks in each partition are counted differently. This is because we've chosen a first, second and third block instead of simply dividing $\underline{6}$ into three blocks of size two.

How can we solve the dilemma? Actually, the discussion of what went wrong contains the key to the solution: The multinomial coefficient counts ordered collections of k blocks and we want unordered collections. Since the blocks in a partition are all distinct, there are $k!$ ways to order the blocks and so the multinomial coefficient counts each unordered collection $k!$ times. Thus we must simply divide the multinomial coefficient by $k!$.

If this dividing by $k!$ bothers you, try looking at it this way. Let $f(k)$ be the number of ways to carry out Step 2. Since the k blocks can be permuted in $k!$ ways, the Rule of Product tells us that there are $f(k)\,k!$ ways to select k ordered blocks of 2 elements each. Thus $f(k)\,k! = \binom{2k}{2,\ldots,2}$.

Since there is just one way to carry out Step 3, the Rule of Product tells us that the number of involutions is

$$\binom{n}{2k}\,\frac{1}{k!}\,\binom{2k}{2,\ldots,2} \;=\; \frac{n!}{(2k)!(n-2k)!}\,\frac{1}{k!}\,\frac{(2k)!}{(2!)^k}.$$

Simplifying and using the Rule of Sum to combine the various possible values of k, we obtain

Theorem 2.2 *The number of involutions of $\underline{n}$ is*

$$\sum_{k=0}^{\lfloor n/2\rfloor} \frac{n!}{(n-2k)!2^k k!}.$$

The notation $\lfloor x\rfloor$ stands for the *largest integer* in x; that is, the largest integer $m \le x$. It is also called the *floor* of x. □

***Example 2.10 Permutation matrices and parity** This example assumes some familiarity with matrices and determinants.

Suppose f and g are permutations of $\underline{n}$. We can define an $n \times n$ matrix F to consist of zeroes except that the (i,j)th entry, $F_{i,j}$, equals one whenever $f(j) = i$. Define G similarly. Then

$$(FG)_{i,j} \;=\; \sum_{k=1}^{n} F_{i,k}G_{k,j} \;=\; F_{i,g(j)},$$

since $G_{k,j} = 0$ except when $g(j) = k$. By the definition of F, this entry of F is zero unless $f(g(j)) = i$. Thus $(FG)_{i,j}$ is zero unless $(fg)(j) = i$, in which case it is one. We have proven that FG corresponds to fg. In other words:

Composition of permutations corresponds to multiplication of matrices.

It is also easy to prove that f^{-1} corresponds to F^{-1}. For example, the permutations $f = (1,3,2)(4)$ and $g = (1,2,3,4)$, written in cycle form, correspond to

$$F \;=\; \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad G \;=\; \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad \text{while} \quad FG \;=\; \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix},$$

which corresponds to the cycle form $(1)(2)(3,4)$, which equals fg.

Using this correspondence, we can prove things such as $(fg)^{-1} = g^{-1}f^{-1}$ and $(f^k)^{-1} = (f^{-1})^k$ by noting that they are true for matrices F and G.

Note that, since the matrix F contains exactly one 1 in each row and column, its determinant is either $+1$ or -1.

Definition 2.4 Parity of a permutation *If the matrix F corresponds to the permutation f, we call $\det F$ the* **parity** *of f and write it $\chi(f)$. If $\chi(f) = +1$, we say f is* **even** *and, if $\chi(f) = -1$, we say that f is* **odd**.

The rest of this example is devoted to proving:

Theorem 2.3 Properties of parity *Suppose f and g are permutations of $\underline{n}$. The following are true.*

(a) $\chi(fg) = \chi(f)\,\chi(g)$.

(b) *If g is f with the elements of $\underline{n}$ relabeled, then $\chi(f) = \chi(g)$.*

(c) *If f is written a product of (not necessarily disjoint) cycles, then $\chi(f) = (-1)^k$, where k is the number of cycles in the product that have even length. In particular, if f is written as a product of k 2-cycles, then $\chi(f) = (-1)^k$.*

(d) *For $n > 1$, exactly half the permutations of $\underline{n}$ are even and exactly half are odd.*

Let F and G be the matrices corresponding to f and g. Then (a) follows from $\det(FG) = \det(F)\,\det(G)$.

In the next paragraph, we'll show that a relabelling can be written $G = P^tFP$ where P is permutation matrix and P^t is the transpose of P. Take determinants:

$$\chi(g) \;=\; \det G \;=\; \det(P^tFP) \;=\; \det P^t\,\det F\,\det P.$$

Now $\det P^t = \det P$ for any matrix P and $(\det P)^2 = \det P$ since the determinant of a permutation matrix is $+1$ or -1. Thus we have $\chi(g) \;=\; \det F \;=\; \chi(f)$.

We must prove the relabeling claim. To relabel, we must permute the columns of F in some fashion and permute the rows of F in the same way. You should convince yourself that multiplying a matrix M by a permutation matrix P gives a matrix MP in which the columns of M have been permuted. Permuting the rows of F is the same as permuting the columns of F^t, except that the resulting matrix is transposed. Thus $(F^tP)^t$ permutes the rows of F and so the rows and columns are permuted by $(F^tP)^tP \;=\; P^tFP$.

If $f = c_1 \cdots c_k$, then $\chi(f) = \chi(c_1) \cdots \chi(c_k)$. Thus (c) follows if we know the parity of a single cycle c. With appropriate relabeling, the matrix corresponding to an m-cycle c is the $m \times m$ matrix

$$C_m \;=\; \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & 0 & \cdots & 0 \end{pmatrix}$$

Expanding about the first row, we have $\det C_m = -\det C_{m-1}$. Since $\det C_2 = -1$, it follows that $\det C_m = (-1)^{m-1}$. Thus c_m is odd if and only if m is even. This completes the proof of (c).

Let $\mathcal{P}_o$ and $\mathcal{P}_e$ be the sets of odd and even permutations of $\underline{n}$, respectively. Since $n \geq 2$, the permutation $t = (1,2)$ is among the permutations. Since $\chi(t) = -1$, it follows from (c) that $T_o : f \mapsto tf$ is map from $\mathcal{P}_o$ to $\mathcal{P}_e$. Since t^2 is the identity function, $T_e : g \mapsto tg$ is the inverse of T_o. You should be able to see that this implies that T_o and T_e are bijections and so $\mathcal{P}_o = \mathcal{P}_e$, proving (d).

For those familiar with group theory, the set of permutations of $\underline{n}$ is called the symmetric group on n symbols and the subset of even permutations is called the alternating group on n symbols. □

Exercises

2.2.1. This exercise lets you check your understanding of cycle form. The answers are given at the end of this problem set. A permutation is given in one-line, two-line or cycle form. Convert it to the other two forms. Give its inverse in all three forms. (The answers are given at the end of this problem set.)

(a) (1,5,7,8) (2,3) (4) (6).

(b) $\left(\begin{smallmatrix}1&2&3&4&5&6&7&8\\8&3&7&2&6&4&5&1\end{smallmatrix}\right)$.

(c) (5,4,3,2,1), which is in one-line form.

(d) (5,4,3,2,1), which is in cycle form.

2.2.2. Let f be a permutation of $\underline{n}$. The cycle of f that contains 1 is called the *cycle generated by* 1.

(a) Prove that the number of permutations in which the cycle generated by 1 has length n is $(n-1)!$.

(b) For $1 \le k \le n$, prove that the number of permutations in which the cycle generated by 1 has length k is $(n-1)!$, independent of the value of k. (Remember that a permutation must permute *all* of $\underline{n}$.)

(c) Conclude that if $1 \le k \le n$ and a permutation of $\underline{n}$ is selected uniformly at random, then the probability that 1 belongs to a k-cycle is $1/n$, independent of k.

2.2.3. A carnival barker has four cups upside down in a row in front of him. He places a pea under the cup in the first position. He quickly interchanges the cups in the first and third positions, then the cups in the first and fourth positions and then the cups in the second and third positions. This entire set of interchanges is done a total of five times. Where is the pea?
Hint. Write one entire set of interchanges as a permutation in cycle form.

2.2.4. Let $P_k(n)$ be the number of permutation of $\underline{n}$ all of whose cycles have length at most k. Thus $P_1(n) = 1$ since only the identity permutation has all cycles of length 1. Also, $P_2(n)$ is the number of involutions. For later convenience, define $P_k(0) = 1$.

(a) By considering the cycle containing $n+1$, prove that $P_2(n+1) = P_2(n) + nP_2(n-1)$ for $n \ge 0$

(b) State and prove a similar recursion for P_3.

2.2.5. A *fixed point* of a permutation f is an element x of the domain such that $f(x) = x$. A *derangement* is a permutation f with no fixed points; i.e., $f(x) \ne x$ for all x.

(a) Prove that the probability that a random permutation f of $\underline{n}$ has $f(k) = k$ equals $1/n$.

(b) If we treat the n events $f(1) \ne 1, \ldots, f(n) \ne n$ as independent, what is the probability that f is a derangement? Conclude that we might expect approximately $n!/e$ derangements of $\underline{n}$. In Example 4.5 (p. 99), you'll see that this heuristic estimate is extremely accurate.

(c) Let D_n be the number of derangements of $\underline{n}$. Prove that the number of permutations of $\underline{n}$ with exactly k fixed points is $D_{n-k}\binom{n}{k} \approx \frac{n!}{k!\,e}$.

2.2.6. Let $z(n,k)$ be the number of permutations of $\underline{n}$ having exactly k cycles. These are called the *signless Stirling numbers of the first kind*. Our notation is not standard. The notation $s(n,k)$ is commonly used both for these and for the Stirling numbers of the first kind, which may differ from them in sign.

(a) Prove the recursion

$$z(n+1,k) = \sum_{i=0}^{n} \binom{n}{i} i!\; z(n-i,k-1).$$

(b) Give initial conditions and the range of n and k for which the recursion is valid.

(c) Construct a table of $z(n,k)$ for $n \le 5$. Note: You can obtain a partial check on your calculations by using $\sum_{k\ge 0} z(n,k) = n!$.

2.2.7. We want to compute the average of $|a_1 - a_2| + |a_2 - a_3| + \cdots + |a_{n-1} - a_n|$ over all permutations $(a_1, \ldots, a_n)$ of $\underline{n}$.

(a) Show that the average equals

$$\frac{2}{n} \sum_{i \le i \le j \le n} (j - i).$$

(b) Show that the answer is $\frac{n^2-1}{2}$.

Answers

2.2.1. (a) $\left(\begin{smallmatrix} 1&2&3&4&5&6&7&8 \\ 5&3&2&4&7&6&8&1 \end{smallmatrix}\right)$ is the two line form and (5,3,2,4,7,6,8,1) is the one-line form. (We'll omit the two-line form in the future since it is simply the one line form with $1, 2, \ldots$ placed above it.) The inverse is (1,8,7,5) (2,3) (4) (6) in cycle form and (8,3,2,4,1,6,5,7) in one-line form.

(b) The cycle form is (1,8) (2,3,7,5,6,4). The inverse in cycle form is (1,8) (2,4,6,5,7,3) and in one-line form is (8,4,2,6,7,5,3,1).

(c) The cycle form is (1,5) (2,4) (3). The permutation is its own inverse.

(d) This is not the standard form for cycle form. Standard form is (1,5,4,3,2). The one-line form is (5,1,2,3,4). The inverse is (1,2,3,4,5) in cycle form and (2,3,4,5,1) in one line form.

2.3 Other Combinatorial Aspects of Functions

Monotonic Functions and Unordered Lists

The one-line notation for a function is simply an (ordered) list, so there is a simple correspondence (i.e., bijection) between lists and functions: A k-list from S is a function $f: \underline{k} \to S$. If f is an injection, the list has no repeats.

How can we get unordered lists to correspond to functions? (Recall that unordered lists correspond to sets or multisets depending on whether repeats are forbidden or not.) The secret is a two step process. First, think of a unique way to order the list, say $s_1, s_2, \ldots, s_k$. Second, interpret the resulting list as a one-line function as done above. In mathematics, people refer to a unique thing (or process or whatever) that has been selected as *canonical*. Thus one would probably speak of a canonical ordering of the list rather than a unique ordering; however, both terms are correct.

Let's look at a small example. Here's a listing of all $\binom{5+3-1}{3} = 35$ of the unordered 3-lists with repetition from $\underline{5}$. In the listing, an entry like 2,5,5 stands for the 3-list containing one 2 and two 5's.

1,1,1	1,1,2	1,1,3	1,1,4	1,1,5	1,2,2	1,2,3	1,2,4	1,2,5	1,3,3	1,3,4	1,3,5
1,4,4	1,4,5	1,5,5	2,2,2	2,2,3	2,2,4	2,2,5	2,3,3	2,3,4	2,3,5	2,4,4	2,4,5
2,5,5	3,3,3	3,3,4	3,3,5	3,4,4	3,4,5	3,5,5	4,4,4	4,4,5	4,5,5	5,5,5	

We've simply arranged the elements in each of the 3-lists to be in "nondecreasing order." Let $b_1, b_2, \ldots, b_k$ be an ordered list. We say the list is in *nondecreasing order* if the values are not decreasing as we move from one element to the next; that is, if $b_1 \le b_2 \le \cdots \le b_k$. We say that $f \in \underline{n}^{\underline{k}}$ is a *nondecreasing function* if its one-line form is nondecreasing; that is, $f(1) \le f(2) \le \cdots \le f(k)$. The list of lists we've created is a bijection between (i) the unordered 3-lists with repetition from $\underline{5}$ and (ii) the nondecreasing functions in $\underline{5}^{\underline{3}}$ written in one-line notation. Thus, 3-multisets of $\underline{5}$ correspond to nondecreasing functions.

In a similar fashion we say that

$$\text{the list } b_1, b_2, \ldots, b_k \text{ is in } \left\{ \begin{array}{c} \textit{nonincreasing} \\ \textit{decreasing} \\ \textit{increasing} \end{array} \right\} \textit{ order if } \left\{ \begin{array}{l} b_1 \ge b_2 \ge \cdots \ge b_k; \\ b_1 > b_2 > \cdots > b_k; \\ b_1 < b_2 < \cdots < b_k. \end{array} \right.$$

Again, this leads to similar terminology for functions. All such functions are also called *monotone functions*. Some people say "strictly decreasing" when we say "decreasing," and likewise for (strictly) increasing. This is a good practice because it helps avoid confusion, therefore we'll usually do it. Also, people say *weakly decreasing* instead of nonincreasing, a convention we often adopt.

We can interchange the strings "decreas" and "increas" in the previous paragraphs and read the functions in the list backwards. Do it. In summary:

$$\left.\begin{array}{rcl} \text{nonincreasing} & = & \text{weakly decreasing} \\ \text{decreasing} & = & \text{strictly decreasing} \\ \text{nondecreasing} & = & \text{weakly increasing} \\ \text{increasing} & = & \text{strictly increasing} \end{array}\right\} \text{ means } \left\{\begin{array}{l} b_1 \geq b_2 \geq \cdots \geq b_k; \\ b_1 > b_2 > \cdots > b_k; \\ b_1 \leq b_2 \leq \cdots \leq b_k; \\ b_1 < b_2 < \cdots < b_k. \end{array}\right.$$

In the bijection we gave from 3-lists to functions, the 3-lists without repetition correspond to the strictly increasing functions. For example, 1,2,3 and 1,3,4 correspond to strictly increasing functions written in one-line notation. Thus 3-subsets of $\underline{5}$ correspond to strictly increasing functions. The bijection between 3-subsets of $\underline{5}$ and strictly decreasing functions is given by

$$3,2,1 \quad 4,2,1 \quad 4,3,1 \quad 4,3,2 \quad 5,2,1 \quad 5,3,1 \quad 5,3,2 \quad 5,4,1 \quad 5,4,2 \quad 5,4,3.$$

The function $(4,3,1)$ corresponds to the 3-subset $\{4,3,1\} = \{1,3,4\}$ of $\underline{5}$.

All these things are special cases of

Theorem 2.4 *There is a bijection between unordered k-lists with repetition made from $\underline{n}$ and the weakly increasing (resp. weakly decreasing) functions in $\underline{n}^{\underline{k}}$. In this correspondence, lists without repetition (i.e., sets) correspond to the strictly increasing (resp. strictly decreasing) functions.*

Example 2.11 A bijection between strict and nonstrict functions Let $m = n+k-1$. We will construct a bijection between the weakly decreasing functions in $\underline{n}^{\underline{k}}$ and the strictly decreasing functions in $\underline{m}^{\underline{k}}$.

Let $f \in \underline{n}^{\underline{k}}$ be a weakly decreasing function. Define a function g by $g(i) = f(i) + k - i$. This is a strictly decreasing function because

$$g(i) - g(i+1) \;=\; \Big(f(i) + k - i\Big) - \Big(f(i+1) + k - (i+1)\Big) \;=\; 1 + \Big(f(i) - f(i+1)\Big) \;\geq\; 1.$$

It has the same domain, $\underline{k}$, as f, but its codomain is $\underline{n+k-1} = \underline{m}$ because $g(1)$ is $k-1$ larger than $f(1)$ and $f(1)$ may be as large as n. Let's give this map from weakly decreasing functions in $\underline{n}^{\underline{k}}$ to strictly decreasing functions in $\underline{n-k+1}^{\underline{k}}$ the name φ. We will prove that φ is a bijection.

It is easy to see that $f_1 \neq f_2$ implies that $\varphi(f_1) \neq \varphi(f_2)$ and so φ is an injection. We claim it is a bijection. To see this, suppose that $h \in \underline{m}^{\underline{k}}$ is strictly decreasing. Define f by $f(i) = h(i) - (k-i)$. This is a function in $\underline{n}^{\underline{k}}$ with $\varphi(f) = h$. To complete the proof, we must prove that it is weakly decreasing. We have

$$f(i) - f(i+1) \;=\; \Big(h(i) - h(i+1)\Big) - 1 \;\geq\; 1 - 1 \;=\; 0.$$

We can combine our knowledge that φ is a bijection with the Theorem 2.4 to obtain another proof for the formula for the number of k-multisets that can be formed from $\underline{n}$. By Theorem 2.4, the number of k-multisets that can be formed from $\underline{n}$ is the same as the number of weakly decreasing functions from $\underline{k}$ to $\underline{n}$. By φ, the latter equals the number of strictly decreasing functions from $\underline{k}$ to $\underline{m}$. By the theorem, this equals the number of k-subsets of $\underline{m}$. Since this last number is $\binom{m}{k} = \binom{n+k-1}{k}$, there are that many k-multisets that can be formed from $\underline{n}$. This is very similar to a proof that was given after Theorem 1.8 (p. 37). □

Example 2.12 Unimodal sequences A sequence that is first weakly increasing and then weakly decreasing is sometimes called unimodal (a term which is not strictly correct). The weakly increasing or weakly decreasing part may be empty. Here are some examples of unimodal sequences:

$$1,3,3,4,2 \qquad 1,1,1,1 \qquad 1,3,5,5,4,2,2 \qquad 1,2,3,2,1.$$

A variety of counting questions can be asked about unimodal sequences. Generally they can all be handled by the following method:

1. Imagine breaking the sequence into three pieces:

$$\underbrace{a_1 \le a_2 \le \cdots \le a_{k-1}}_{k-1 \text{ items}} \quad < \quad a_k \quad \ge \quad \underbrace{a_{k+1} \ge a_{k+2} \cdots \ge a_\ell}_{\ell-k \text{ items}},$$

   where a_k is the first occurrence of the largest element in the sequence.

2. Obtain a formula for the number of such sequences based on the value of a_k and, possibly, on the values of k and ℓ.

3. Sum the result of the previous step.

We'll do a couple of examples.

How many monotonic sequences are there in which the inequalities are all strict and the elements lie in $\underline{n}$? Suppose the largest element is t. The sequence elements preceding t must be a strict monotonic sequence with all elements in $\underline{t-1}$. Since such sequences correspond to subsets of $\underline{t-1}$ there are 2^{t-1} choices for such a sequence. Similarly, there are 2^{t-1} choices for the elements after t. Thus there are $(2^{t-1})^2$ strict monotonic sequences of positive integers with largest element t. Hence the answer to the original question is

$$\sum_{t=1}^{n}(2^{t-1})^2 \;=\; 1+4+16+\cdots+4^{n-1} \;=\; \frac{1-4^n}{1-4} \;=\; \frac{4^n-1}{3}.$$

How many weakly monotonic sequences of length ℓ are there whose elements lie in $\underline{n}$? Let $a_k = t$.

- The weakly increasing sequence preceding a_k corresponds to a $(k-1)$-multiset formed from $\underline{t-1}$. There are $\binom{t+k-3}{k-1}$ of them. We have to be careful here because of the case $t = 1$.
- It turns out to be easier to treat the case $t = 1$ separately. There is only one weakly monotonic sequence of length ℓ with largest term 1, namely the sequence of all ones.
- The weakly decreasing sequence following a_k corresponds to a $(\ell-k)$-multiset formed from $\underline{t}$. There are $\binom{\ell+t-k-1}{\ell-k}$ of them.

Thus the number of weakly monotonic sequences of length ℓ whose elements lie in $\underline{n}$ is

$$1+\sum_{t=2}^{n}\sum_{k=1}^{\ell}\binom{t+k-3}{k-1}\binom{\ell+t-k-1}{\ell-k}. \;\square$$

Image and Coimage

Again, let $f: A \to B$ be a function. The *image* of f is the set of values f actually takes on: $\text{Image}(f) = \{\, f(a) \mid a \in A \,\}$. The definition of a surjection can be rewritten $\text{Image}(f) = B$.*

For each $b \in B$, the *inverse image* of b, written $f^{-1}(b)$ is the set of those elements in A whose image is b; i.e.,

$$f^{-1}(b) \;=\; \{\, a \mid a \in A \text{ and } f(a) = b \,\}.$$

This extends our earlier definition of f^{-1} from bijections to all functions; however, such an f^{-1} can't be thought of as a function from B to A unless f is a bijection because it will not give a unique $a \in A$ for each $b \in B$. (There is a slight abuse of notation here: If $f: A \to B$ is a bijection, our new notation is $f^{-1}(b) = \{a\}$ and our old notation is $f^{-1}(b) = a$.)

The collection of nonempty inverse images of elements of B is called the *coimage* of f.

We claim that the coimage of f is the partition of A whose blocks are the maximal subsets of A on which f is constant. For example, if $f \in \{a, b, c\}^{\underline{5}}$ is given in one-line form as (a, c, a, a, c), then

$$\text{Coimage}(f) \;=\; \{f^{-1}(a), f^{-1}(c)\} \;=\; \big\{\{1, 3, 4\}, \{2, 5\}\big\},$$

f is a on $\{1, 3, 4\}$ and is c on $\{2, 5\}$.

We now prove the claim. If $x \in A$, let $y = f(x)$. Then $x \in f^{-1}(y)$ and so the union of the nonempty inverse images contains A. Clearly it does not contain anything which is not in A. If $y_1 \neq y_2$, then we cannot have $x \in f^{-1}(y_1)$ and $x \in f^{-1}(y_2)$ because this would imply $f(x) = y_1$ and $f(x) = y_2$, a contradiction. Thus $\text{Coimage}(f)$ is a partition of A. Clearly x_1 and x_2 belong to the same block if and only if $f(x_1) = f(x_2)$. Hence a block is a maximal set on which f is constant.

Example 2.13 Blocks and Stirling numbers How many functions in B^A have a coimage with exactly k blocks?

This means that the coimage is a partition of A having exactly k blocks. Recall that $S(n, k)$, a Stirling number of the second kind, denotes the number of partitions of an n-set into k blocks. (See Example 1.27 (p. 34).) There are $S(|A|, k)$ ways to choose the blocks of the coimage. The partition of A does not fully specify a function $f \in B^A$. To complete the specification, we must specify the image of the elements in each block, in other words, an injection from the set of blocks to B. This is an ordered selection of size k without replacement from B. There are $|B|!/(|B| - k)!$ such injections, independent of which k block partition of A we are considering. By the Rule of Product, there are $S(|A|, k)\big(|B|!/(|B| - k)!\big)$ functions $f \in B^A$ with $|\,\text{Coimage}(f)| = k$. □

We can describe the image and coimage of a function by the arrow pictures introduced at the end of Example 2.4. $\text{Image}(f)$ is the set of those $b \in B$ which appear as labels of arrowheads. A block in $\text{Coimage}(f)$ is the set of labels on the tails of those arrows that all have their heads pointing to the same value; for example, the block of $\text{Coimage}(f)$ arising from $b \in \text{Image}(f)$ is the set of labels on the tails of those arrows pointing to b.

* The image is also called the range. Unfortunately, the codomain is also sometimes called the range.

Example 2.14 Blocks of given sizes Let $\vec{b}$ be a vector of nonnegative integers such that $\sum_{k\geq 1} kb_k = n$, and let $B(n,\vec{b})$ be the number of partitions of $\underline{n}$ having exactly b_k blocks of size k. We call $\vec{b}$ the *type* of a partition and so $B(n,\vec{b})$ is the number of partitions of type $\vec{b}$.

Consider the possible coimages of functions $f\colon \underline{n} \to \underline{m}$. Since the coimage is a partition of $\underline{n}$, we can talk about the type of the coimage, too. There is a restriction: Since the coimage can't have more blocks than the size of the codomain, $b_1 + 2b_2 + \cdots \leq m$.

We want to compute $B(n,\vec{b})$. To do this, we first partition $\underline{n}$ into ordered blocks where there are kb_k elements in block k for each k. Next, for each k, we partition block k into b_k blocks each of size k. For the first step, all we need is a multinomial coefficient since that step is precisely the multinomial coefficient setup. The second step would also be a multinomial coefficient setup if we had put the blocks of size k in an ordered list. Since there are $b_k!$ ways to order the b_k blocks of size k, we need to divide that multinomial coefficient by $b_k!$. This gives us

$$B(n,\vec{b}) \;=\; \binom{n}{b_1, 2b_2, 3b_3, \ldots} \prod_{b_k\neq 0} \frac{1}{b_k!} \binom{kb_k}{\underbrace{k,k,\ldots,k}_{b_k \text{ copies of } k}} \;=\; \frac{n!}{\prod_{b_k\neq 0} b_k!\,(k!)^{b_k}}. \qquad \square$$

The Pigeonhole Principle

The *Pigeonhole Principle* is a method for obtaining statements of the form

$$\text{If } n \geq g(d), \;\; \text{then } \mathcal{A}(n,d),$$

where $\mathcal{A}$ is a statement and g is some function depending on $\mathcal{A}$. For example, we'll see that, if $n \geq d^2+1$, then any sequence of n distinct numbers contains a monotonic subsequence of length d. (What follows "then" is $\mathcal{A}(n,d)$.)

Here is a statement of the principle in two forms.

Theorem 2.5 Pigeonhole Principle Function form: *Suppose A and B are sets with $|A| > |B|$, then for every function $f : A \to B$ there is a $b \in B$ with $|f^{-1}(b)| > 1$.*
Partition form: *Suppose $\mathcal{P}$ is a partition of the set A into less than $|A|$ blocks. Then some block contains more than one element of A.*

You should be able to prove this theorem. In fact, it is so simple we should probably not even call it a theorem. To see why the two forms of the theorem are equivalent, first suppose $f : A \to B$. Let $\mathcal{P}$ be the coimage of f. It must have at most $|B| < |A|$ blocks. Conversely suppose $\mathcal{P}$ is a partition of A. Number the blocks in some fashion from 1 to $|\mathcal{P}|$. Let $B = \{1,\ldots,|\mathcal{P}|\}$ and define $f : A \to B$ by letting $f(a)$ be the number of the block that contains A.

Where did the rather strange name "Pigeonhole Principle" come from? Old style desks often had what looked like a stacked array of boxes that were open in the front. These boxes were usually used to hold various letters and folded or rolled papers. The boxes were called pigeon holes because of their resemblance to nesting boxes in pigeon coops. If $|A|$ letters are placed in $|B|$ pigeonholes in a desk and $|A| > |B|$, then at least one pigeonhole contains at least two documents.

Example 2.15 Subset sums Given a set P of integers, let $\sum P$ be the sum of the elements of P. We say that a set S of positive integers has the *two-sum property* if there are subsets P and Q of S with $P \neq Q$ and $\sum P = \sum Q$. Not all sets have the two-sum property. For example, $\{1, 2, 4, 8\}$ does not, but $\{1, 2, 4, 5\}$ has the two-sum property—take $P = \{1, 4\}$ and $Q = \{5\}$. The set $S = \{1, 2, 4, 8\}$ fails because its elements grow too rapidly. Can we put some condition on $|S|$ and $\max S$, the largest element of S, so that S must have the two sum property? Since n can't be too large, we might expect a statement of the form

$$\text{If } |S| \geq k \text{ and } \max S \leq h(k), \text{ then } S \text{ has the two-sum property.} \qquad 2.1$$

If a subset of S has the two-sum property, then so does S. Hence it suffices to prove (2.1) when $|S| = k$ since any S with $|S| > k$ can replaced by a subset S' of size k. Thus (2.1) is equivalent to

$$\text{If } |S| = k \text{ and } \max S \leq h(k), \text{ then } S \text{ has the two-sum property.}$$

Since we want to look for repeated values of subset sums, our function f will map subsets to their sums. Thus $A = 2^S$, the subsets of S, and $f : A \to \{0, 1, \ldots, \sum S\}$ is defined by $f(P) = \sum P$.

To apply the pigeonhole principle we need the cardinality of the domain and codomain of f. We have $|A| = 2^{|S|} = 2^k$. Since $B = \{0, 1, \ldots, \sum S\}$, we have $|B| = \sum S + 1$. For the Pigeonhole Principle, we need $2^k > \sum S + 1$. To get this, we need an upper bound for $\sum S$. Let $\max S = n$ and notice that

$$\sum S \;\leq\; \underbrace{n + (n-1) + (n-2) + \cdots + (n-k+1)}_{k \text{ terms}} \;\leq\; nk.$$

Thus it suffices to have $2^k \geq nk + 1$. This is easily solved to get an inequality for n. Recalling that $n = \max S$ and $k = |S|$, we have proved that any set S of positive integers has the two-sum property if

$$\max S \;\leq\; \frac{2^{|S|} - 1}{|S|}. \quad \square$$

Example 2.16 Monotonic subsequences We will prove

Theorem 2.6 *Given a sequence of more than mn distinct numbers, there must be either a subsequence of $n + 1$ decreasing numbers or a subsequence of $m + 1$ increasing numbers.*

A subsequence need not be consecutive; for example, 1,2,3 is a subsequence of 7,1,6,4,2,4,3.

The theorem is best possible because the following sequence contains nm numbers, the longest decreasing subsequence has length n and the longest increasing subsequence has length m.

$$n, (n-1), \ldots, 1, \quad 2n, (2n-1), \ldots, n+1, \quad 3n, (3n-1), \ldots, 2n+1, \quad \ldots, \quad mn, (mn-1), \ldots, (m-1)n+1. \qquad 2.2$$

How can we prove the theorem? Here is a fairly natural approach which turns out to be incorrect. We could try to "grow" sequences as follows. Let $a_1, \ldots a_\ell$ be the given sequence. Then a_ℓ is both an increasing and a decreasing sequence starting at ℓ. We back up from ℓ one step at a time until we reach 1. Suppose we have decreasing and increasing sequences starting at t. If $a_{t-1} < a_t$, we can put a_{t-1} at the front of our increasing sequence and increase its length by one. If $a_{t-1} > a_t$ we can put a_{t-1} at the front of our decreasing sequence and increase its length by one. Each step increases one length or the other by 1. Thus, when we reach a_1, the sum of the lengths is the initial sum plus the number of steps: $2 + (\ell - 1) = \ell + 1$. Thus, if $\ell \geq m + n$, we have either an increasing subsequence longer than m or a decreasing one longer than n. This can't be right—it's better than the claim in the theorem and (2.2) tells us that's best possible. Can you see what is wrong?

* * * Stop and think about this! * * *

We must compare a_{t-1} with the starts of the increasing and decreasing subsequences that we've built so far and, when $t < \ell$ only one of these subsequences begins with a_t.

Can we salvage anything from what we did? Suppose no decreasing subsequence is longer than n and no increasing subsequence is longer than m. Let I_t be the length of the longest increasing subsequence that starts with a_t and let D_t be the length of the longest decreasing subsequence that starts with a_t. Define a map $f : \underline{\ell} \to \underline{m} \times \underline{n}$ by $f(t) = (I_t, D_t)$. We'll show that f is an injection. By the Pigeonhole Principle, we cannot have $|A| > |B|$. In other words $\ell \leq m \times n$. The contrapositive is the theorem because it says that if $\ell > m \times n$, then it is not true that both (a) no increasing subsequence exceeds m and (b) no decreasing subsequence exceeds n. In other words, if $\ell > m \times n$, there is either an increasing subsequence longer than m or a decreasing subsequence longer than n.

It remains to show that f is an injection. Suppose $i < j$. You should be able to see that

$$\text{If } a_i < a_j, \text{ we have } I_i > I_j.$$
$$\text{If } a_i > a_j, \text{ we have } D_i > D_j.$$

This completes the proof. Another proof is given in the exercises. □

Exercises

2.3.1. This exercise lets you check your understanding of the definitions. In each case below, some information about a function is given to you. Answer the following questions and give reasons for your answers: (The answers are given at the end of this problem set.)

(i) Have you been given enough information to specify the function; i.e., would this be enough data for a function envelope?

(ii) Can you tell whether or not the function is an injection? a surjection? a bijection? If so, what is it?

(a) $f \in \underline{4}^{\underline{5}}$, $\text{Coimage}(f) = \{\{1,3,5\},\{2,4\}\}$.

(b) $f \in \underline{5}^{\underline{5}}$, $\text{Coimage}(f) = \{\{1\},\{2\},\{3\},\{4\},\{5\}\}$.

(c) $f \in \underline{4}^{\underline{5}}$, $f^{-1}(2) = \{1,3,5\}$, $f^{-1}(4) = \{2,4\}\}$.

(d) $f \in \underline{4}^{\underline{5}}$, $\big|\text{Image}(f)\big| = 4$.

(e) $f \in \underline{4}^{\underline{5}}$, $\big|\text{Image}(f)\big| = 5$.

(f) $f \in \underline{4}^{\underline{5}}$, $\big|\text{Coimage}(f)\big| = 5$.

2.3.2. Let A and B be finite sets and let $f\colon A \to B$ be a function. Prove the following claims.

(a) $|\,\text{Image}(f)| = |\,\text{Coimage}(f)|$.

(b) f is an injection if and only if $|\,\text{Image}(f)| = |A|$.

(c) f is a surjection if and only if $|\,\text{Coimage}(f)| = |B|$.

2.3.3. Let $\vec{b}$ be a vector of nonnegative integers such that $\sum_{k\geq 1} kb_k = n$, let $C(n,\vec{b})$ be the number of permutations of $\underline{n}$ having exactly b_k cycles of length k, and let $B(n,\vec{b})$ be the number of partitions of $\underline{n}$ having exactly b_k blocks of size k. Prove that

$$C(n,\vec{b}) \;=\; B(n,\vec{b}) \prod_{k\geq 1} (k-1)!^{b_k}.$$

2.3.4. How many strictly unimodal sequences are there in which the largest element is in the exact middle of the sequence and no element exceeds n?

2.3.5. How can we get a bijection between partitions of a set and a reasonable class of functions? Does the notion of coimage help? What about partitions with exactly k blocks? This exercise deals with these issues.

(a) Given a partition of $\underline{n}$, let block 1 be the block containing 1. If the first k blocks have been defined, let block $k+1$ be the remaining block that contains the smallest remaining element. This numbers the blocks of a set partition. Number the blocks of the partition

$$\{3,9,2\},\ \{4,1,6\}\ \{5\}\ \{7,8\}.$$

(b) Given a partition B of a $\underline{n}$, associate with it a function $f \in \underline{n}^{\underline{n}}$ as follows. Number the blocks of B as above. Let $f(i)$ be the number of the block of B that contains i. Prove that if two partitions are different, then the functions associated with them are different.

(c) What is the coimage of the function associated with a partition B of $\underline{n}$? (Phrase your answer as simply as possible in terms of B.)

(d) Call a function $f \in \underline{n}^{\underline{n}}$ a *restricted growth function* if $f(1) = 1$ and $f(i) - 1$ is at most the maximum of $f(k)$ over all $k < i$. Which of the following functions in one-line form are restricted growth functions? Give reasons for your answers.

$$2,2,3,3 \quad 1,2,3,3,2,1 \quad 1,1,1,3,3 \quad 1,2,3,1.$$

(e) Prove that a function is associated with a partition of $\underline{n}$ if and only if it is a restricted growth function. (Since you already proved that different partitions have different functions associated with them, this is the desired bijection.)

(f) For $\underline{4}$, list in lexicographic order all restricted growth functions and, for each function, give the partition of $\underline{4}$ that corresponds to it.

2.3.6. Prove the generalized Pigeonhole Principle: Suppose that a set S is partitioned into k blocks. Prove that some block must have more than $(|S|-1)/k$ elements.

2.3.7. Let $f : A \to B$. Use the generalized Pigeonhole Principle to obtain a lower bound on the size of the largest block in the coimage of f.

2.3.8. Suppose $n+1$ numbers are selected from $\underline{2n}$. Prove that we must have selected two numbers such that one is a multiple of the other.
Hint. Write each number in the form $2^k m$ where m is odd. Study occurrences of values of m using the Pigeonhole Principle.

2.3.9. Prove Theorem 2.6 using the generalized Pigeonhole Principle (Exercise 2.3.6) as follows. Given $a_1, \ldots, a_\ell$, define $f : \underline{\ell} \to B$ by letting $f(t)$ be the length of the longest decreasing subsequence starting with a_t. If $i < j < \cdots$ and $f(i) = f(j) = \cdots$, look at the subsequence $a_i, a_j, \ldots$.

2.3.10. Given N, how large must t be so that every set S containing at least t positive integers satisfies the following condition? There are elements $a, b, c, d \in S$ such that (i) $a+b$ and $c+d$ have the same remainder when divided by N and (ii) $\{a,b\} \neq \{c,d\}$.
Hint. Look at remainders when sums of pairs are divided by N.

*2.3.11. Given a set S of integers, prove that the elements of some nonempty subset of S sum to a multiple of $|S|$.

Answers

2.3.1. (a) The domain and codomain of f are specified and f takes on exactly two distinct values. f is not an injection. Since we don't know the values f takes, f is not completely specified; however, it cannot be a surjection because it would have to take on all four values in its codomain.

(b) Since each block in the coimage has just one element, f is an injection. Since $|\operatorname{Coimage}(f)| = 5 = |\text{codomain of } f|$, f is a surjection. Thus f is a bijection and, since the codomain and domain are the same, f is a permutation. In spite of all this, we don't know the function; for example, we don't know $f(1)$, but only that it differs from all other values of f.

(c) We know the domain and codomain of f. From $f^{-1}(2)$ and $f^{-1}(4)$, we can determine the values f takes on the union $f^{-1}(2) \cup f^{-1}(4) = \underline{5}$. Thus we know f completely. It is neither a surjection nor an injection.

(d) This function is a surjection, cannot be an injection and has no values specified.

(e) This specification is nonsense. Since the image is a subset of the codomain, it cannot have more than four elements.

(f) This specification is nonsense. The number of blocks in the coimage of f equals the number of elements in the image of f, which cannot exceed four.

*2.4 Boolean Functions

A *Boolean function* f is a map from $\{0,1\}^n$ to $\{0,1\}$. Thus the domain of f is all n long vectors of zeroes and ones. Boolean functions arise in logic, where 0 is often replaced by F for "False" and 1 by T for "True." Boolean functions also arise in arithmetic, where 0 and 1 are digits of numbers in binary representation. Mathematically, there is no difference between these interpretations; however, the two different interpretations have slightly different notation associated with them.

Example 2.17 Basic Boolean functions Here are three functions from $\{0,1\}^2$ to $\{0,1\}$ in two-line form

$$p = \begin{pmatrix} (0,0) & (0,1) & (1,0) & (1,1) \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad s = \begin{pmatrix} (0,0) & (0,1) & (1,0) & (1,1) \\ 0 & 1 & 1 & 1 \end{pmatrix} \qquad f = \begin{pmatrix} (0,0) & (0,1) & (1,0) & (1,1) \\ 0 & 1 & 1 & 0 \end{pmatrix}.$$

If we think of x and y as integers, we can write $p(x,y) = xy$ and, indeed, this is the notation that is commonly used for p. To emphasize the multiplication, we might write $p(x,y) = x \cdot y$. Suppose that X and Y are statements; e.g., X may be the statement "It is cloudy." and Y may be "It is hot." We can build a more complicated statement from these two in many simple ways. One possibility is

"It is cloudy and it is hot."

We could abbreviate this to "X and Y." Let this compound statement be Z. Let $x = 0$ if X is false and $x = 1$ if X is true. Define y and z similarly. (This is the True/False interpretation of 0 and 1 mentioned earlier.) You should be able to see that $z = p(x,y)$ because Z is true if and only if both X and Y are true. Not surprisingly, the function p is called *and* in logic. Logicians sometimes write $p(x,y) = x \wedge y$ instead of $p(x,y) = xy$.

What interpretation does the function $s(x,y)$ have? If we write it as an arithmetic function on integers, it is a bit of a mess, namely, $x + y - xy$. In logic it has a very simple interpretation. Using the notation of the previous paragraph, let Z be the statement "X or Y." Our usual understanding of this is that Z is true if at least one of X and Y is true. This understanding is equivalent to the mathematical statement $s(x,y) = z$ since s is 1 if at least one of its arguments is 1. Logicians call this function *or*. There are two common ways to denote it: $s(x,y) = x \vee y$ and $s(x,y) = x + y$. The $x + y$ notation is a bit unfortunate because *it does not mean that x and y are to be added as integers*. Since the plus sign notation is commonly used in discussion of circuits, we will use it here. Remember:

If $x, y \in \{0,1\}$, then $x + y$ means OR, *not* addition.

How can our third function f be interpreted? In terms of logic, $f(x, y)$ corresponds to a statement which is true when exactly one of X and Y is true. Thus f is called the *exclusive or* and we write it as $f(x, y) = x \oplus y$. This function does not appear often in logic but it is important in binary arithmetic: $x \oplus y$ is the unit's (rightmost) digit of the binary sum of the numbers x and y. (Recall that the binary sum of 1 and 1 is 10.)

We need a little more notation. Negation, $n(x)$, also called complementation, is an important function. It is $n(x) = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$; i.e., $n(x)$ is true if and only if x is false. In terms of the previous notation, we can write $n(x) = x \oplus 1$. The notations x' and $\overline{x}$ are both commonly used for $n(x)$. We will use x'. You should be able to see quickly that $x'' = x$. □

Just as in ordinary algebra, we can combine the functions we introduced in the previous example. Some formulas such as $xy = yx$ and $(x + y)z = xz + yz$, which are true in ordinary arithmetic are true here, too. There are also some surprises such as $xx = x + x = x$ and $x + xy = x$. For those who like terminology, $xx = x$ and $x + x = x$ are called the idempotent laws for multiplication and addition and $x + xy = x$ is called the absorbtion law for addition. How can we check to see if an identity for Boolean expressions is correct? We can do this in one of two ways:

(a) (Brute force) By substituting all possible combinations of zeroes and ones for the variables, we may be able to prove that both sides of the equal sign always take on identical values.

(b) (Algebra) By manipulation using known identities, we may be able to prove that both sides of the equal sign are equal to the same expression.

The brute force method is guaranteed to always work, but the algebraic method is often quicker if you can see how to make the algebra work out. Let's see how these methods work.

To verify $x + x = x$ by brute force, note first that when $x = 0$ the left side is $0 + 0$, which is 0, and second that when $x = 1$ the left side is $1 + 1$, which is 1.

We now use algebra to verify $(x + y)(x + z) = x + yz$ using the identities we mentioned above. We leave it to you to indicate which of the above identities is used at each step.

$$\begin{aligned}
(x + y)(x + z) &= x(x + z) + y(x + z) \\
&= (x + z)x + (x + z)y \\
&= xx + zx + xy + zy \\
&= x + xz + xy + yz \\
&= x + xy + yz \\
&= x + yz.
\end{aligned}$$

Of course, one would not normally write out all the steps. It would probably be shortened to

$$(x + y)(x + z) = x + xy + xz + yz = x + yz,$$

just as we take shortcuts in ordinary algebra.

Example 2.18 Truth tables Two-line notation is a convenient way of proving identities by brute force when we modify it slightly: Instead of listing one function with domain $\{0,1\}^n$ and codomain $\{0,1\}$, we list several. This gives a way of building up various expressions. *Truth tables* are a modification of this: rows and columns are interchanged. Here is the truth table for the basic operations we introduced earlier.

x	y	xy	$x+y$	$x \oplus y$	x'
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Let's use truth tables to prove $(xy)' = x' + y'$. Here's the complete proof (the $(xy)'$ and $x' + y'$ columns are equal):

x	y	xy	$(xy)'$	x'	y'	$x' + y'$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

□

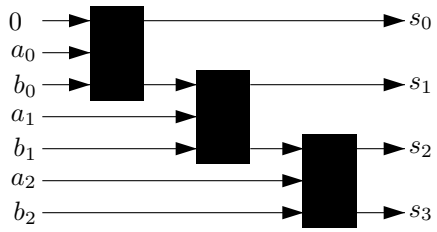
Two important identities that we have not mentioned in full generality are *DeMorgan's laws*:

$$(x \cdot y \cdot \cdots \cdot z)' = x' + y' + \cdots + z', \tag{2.3}$$

$$(x + y + \cdots + z)' = x' \cdot y' \cdot \cdots \cdot z'. \tag{2.4}$$

We prove the first by a modification of brute force and leave the second for you. The product on the left side of (2.3) is 1 if and only if $x = y = \cdots = z = 1$. Thus the left side of (2.3) is 0 if and only if $x = y = \cdots = z = 1$. The right side of (2.3) is 0 if and only if $x' = y' = \cdots = z' = 0$, which is equivalent to $x = y = \cdots = z = 1$. Thus, each side of (2.3) is 0 precisely when $x = y = \cdots = z = 1$. Consequently both sides are 1 for all other values of $x, y, \ldots, z$. This completes the proof.

Example 2.19 An adder One of the basic operations in a computer is addition. We can imagine designing a device to add two numbers the way we learned to do it in school: First add the digits in the unit's position, record the unit's digit of the sum and remember a carry digit (no carry is a carry digit of zero). Second, add the digits in the ten's position and the carry digit, record the unit's digit of the sum as the ten's digit of the answer and remember a carry digit. And so on. Of course, in a computer we work with binary instead of base ten. For one possible design of an adder, we need a device that takes in three binary digits and produces a sum unit's digit and a carry digit. Represent such a device by a black box with inputs on the left, and outputs on the right so that the top output is the sum unit's digit and the bottom output is the carry digit. To add two k digit binary numbers, we can combine k of these boxes; for example, here's how we can add the two 3 digit binary numbers $a_2a_1a_0$ and $b_2b_1b_0$ to get a sum $s_3s_2s_1s_0$:



We need to specify the outputs of our black box as Boolean functions of the inputs. One way to do this is with a truth table. Another way to do it is by writing down the equations. They are shown in Figure 2.1, with x, y and z as inputs and s and c as the sum and carry, respectively. (You

x	y	z	s	c
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\begin{aligned} s &= x \oplus y \oplus z \\ &= x'y'z + x'yz' + xy'z' + xyz \end{aligned}$$

$$c = xy + xz + yz$$

Figure 2.1 The truth table and algebraic representations of a three bit binary adder. The sum of the bits x, y and z is the two bit binary number cs.

should be able to derive the truth table from the definition of s and c, but you may not see how to write down the algebraic equations.)

What form should we use—truth table or algebraic equations? The answer depends on what we plan to do with the function. If we plan to manipulate it, then an algebraic form is usually best. If we plan to use it in some computer aided circuit design program, then whatever form the program requires is best. □

How can we convert a truth table to an algebraic expression? One method is *disjunctive normal form*. A Boolean function is in disjunctive normal form if it is a sum (OR) of products (ANDs) with each factor in each product being either a variable or its complement. For example, the functions $x'y'z+x'yz'+xy'z'+xyz$ and $xy+xz+yz$ in Figure 2.1 are in disjunctive normal form, but $x\oplus y\oplus z$ is not.

In Example 7.3 (p. 200) we'll prove by induction that every Boolean function can be written in disjunctive normal form. We now give a different proof by showing how to convert the truth table for a function into disjunctive normal form. Let the function be $f(x_1, x_2, \ldots, x_n)$ Our disjunctive normal form will contain one term for each row of the truth table in which $f = 1$. Each term will be a product of n factors. If the row says $x_i = 1$, then x_i appears in the product; otherwise, x_i' appears in the product. For example, look at the function s in Figure 2.1. There are four rows in the truth table with $s = 1$. The first such row has $x = y = 0$ and $z = 1$, which gives us the first term, $x'y'z$, in our disjunctive normal form for s.

It is rather easy to prove that the truth table, T', of the resulting disjunctive normal form, f, equals the truth table, T, it was derived from. A row of T' will be 1 if and only if f is 1, which happens if and only if some term of f is 1. On the other hand, a term of f is 1 if and only if $x_1, \ldots, x_n$ take on the values that led to that term. Those values were associated with a row in T for which the function is 1.

Let's return to Figure 2.1 and look at a disjunctive normal form for c. From the truth table we have

$$c = x'yz + xy'z + xyz' + xyz, \tag{2.5}$$

which is not the same as the disjunctive normal form $c = xy + xz + yz$ given earlier. How can this be? There need not be a unique disjunctive normal form for a function. Is one form better than the other? Yes, since it has fewer and simpler terms than (2.5), $c = xy + xz + yz$ requires less hardware to implement as a disjunctive normal form than does (2.5). How can we find the "best" disjunctive normal form? This is a hard question—even defining "best" may be difficult.

There are other algebraic forms besides disjunctive normal form which are important. With current computer chip technology, the most important is "NAND," which is defined by

$$\text{NAND}(x_1, x_2, \ldots, x_n) = (x_1 x_2 \cdots x_n)', \quad n \geq 1;$$

that is, "NOT the AND of the x_i's." "NAND" is a contraction of "NOT AND". By one of DeMorgan's laws,

$$\text{NAND}(x_1, x_2, \ldots, x_n) \;=\; x_1' + x_2' + \cdots + x_n'. \tag{2.6}$$

It is a simple matter to prove that every Boolean function can be built up out of NANDs: We will do this by proving that every disjunctive normal form can be implemented by using NANDs. Let the disjunctive normal form be $f = p_1 + p_2 + \cdots + p_m$, where $p_i = u_{i,1}u_{i,2}\cdots u_{i,k_i}$ and each u is an input variable or its complement. To complete the proof it suffices to note that

$$\begin{aligned} f &= \text{NAND}(p_1', \ldots, p_m') && \text{by (2.6) and } p'' = p; \\ p_i' &= \text{NAND}(u_{i,1}, \ldots, u_{i,k_i}) && \text{by the definition of NAND;} \\ x_t' &= \text{NAND}(x_t). \end{aligned}$$

Thus f is a NAND of NANDs of $u_{i,j}$'s and each $u_{i,j}$ is a variable or its NAND. For example, from Figure 2.1 we have

$$\begin{aligned} c \;&=\; xy + xz + yz \;=\; \text{NAND}\Big(\text{NAND}(x,y), \text{NAND}(x,z), \text{NAND}(y,z)\Big) \\ s \;&=\; x'y'z + \cdots \;=\; \text{NAND}\Big(\text{NAND}(\text{NAND}(x), \text{NAND}(y), z), \ldots\Big). \end{aligned}$$

Does this direct translation of disjunctive normal form give us the simplest representation by NANDs? (Of course, there may be several disjunctive normal forms, so we would presumably start with the "simplest.") As you may suspect, the answer is "No." In fact, the question is not even well posed. One issue that arises immediately is what about repeated expressions; e.g., if x_1' appears several times, must we compute $\text{NAND}(x_1)$ several times, or may we just compute it once and reuse it? We may usually reuse it, but there are sometimes hardware constraints that do not allow us to. This is much like the problem of computing any complicated algebraic function in which some subexpression appears more than once: If you are clever, you will only compute that subexpression once. We'll not pursue the complex problem of representation by NANDs.

Exercises

2.4.1. Using only the identities in the text, prove $x(x + y) = x$ by algebra. (This is the absorbtion law for multiplication.)

2.4.2. Prove that $x \oplus y = x' \oplus y'$.

2.4.3. The distributive law for multiplication, $\cdot$, over addition, $+$, states that $x \cdot (y + z) = x \cdot y + x \cdot z$. State the following distributive laws. If the law is true, prove it. If the law is false, give a counterexample. *Hint.* Truth tables can be used to find proofs and counterexamples.

(a) addition, $+$, over multiplication, $\cdot$;
(b) multiplication, $\cdot$, over exclusive or, $\oplus$;
(c) addition, $+$, over exclusive or, $\oplus$;
(d) exclusive or, $\oplus$ over multiplication, $\cdot$.

2.4.4. Define $\text{NOR}(x, y, \ldots, z) = (x + y + \cdots + z)'$. Prove that every Boolean function can be expressed using NORs.

2.4.5. Write each of the following in disjunctive normal form. Try to obtain as simple a disjunctive normal form as possible.

(a) $(x \oplus y)(x + y)$.
(b) $(x + y) \oplus z$.
(c) $(x + y + z) \oplus z$.
(d) $(xy) \oplus z$.

2.4.6. Let $f(x, y, z)$ equal z unless $x = y$, in which case $f(x, y, z) = x$. Write $f(x, y, z)$ in disjunctive normal form.

2.4.7. Let $f(x, y, z, w)$ equal w unless $x = y = z$, in which case $f(x, y, z, w) = x$. Write $f(x, y, z, w)$ in disjunctive normal form. Try to obtain as simple a form as you can.

2.4.8. Let $f(x, y, z, w)$ equal zw if $x = y$ and $z + w$ otherwise. Write $f(x, y, z, w)$ in disjunctive normal form. Try to obtain as simple a form as you can.

Notes and References

Further discussion of Boolean functions and circuit design at the level of this text can be found in the texts by Gill [1] and McEliece et al. [2].

1. Arthur Gill, *Applied Algebra for the Computer Sciences*, Prentice-Hall (1976).
2. Robert J. McEliece, Robert B. Ash and Carol Ash, *Introduction to Discrete Mathematics*, Random House (1989).

CHAPTER 3

Decision Trees

Introduction

In many situations one needs to make a series of decisions. This leads naturally to a structure called a "decision tree," which we will define shortly. Decision trees provide a geometrical framework for organizing the decisions. The important aspect is the decisions that are made. Everything we do in this chapter could be rewritten to avoid the use of trees; however, trees

- give us a powerful intuitive basis for viewing the problems of this chapter,
- provide a language for discussing the material,
- allow us to view the collection of all decisions in an organized manner.

We'll begin by studying some basic concepts concerning decision trees. Next we'll relate decision trees to the concepts of "ranking" and "unranking," ideas which allow for efficient computer storage of data which is *a priori* not indexed in a simple manner. Finally we'll study decision trees which contain some bad decisions; i.e., decisions that do not lead to solutions.

Decision trees are particularly useful in the "local" study of recursive procedures. In Sections 7.3 (p. 210) and 9.3 (p. 259) we'll apply this idea to various algorithms.

Make sure you're familiar with the concepts in Chapter 2, especially the first section, the definition of a permutation and the bijections in Theorem 2.4 (p. 52).

3.1 Basic Concepts of Decision Trees

Decision trees provide a method for systematically listing a variety of functions. We'll begin by looking at a couple of these. The simplest general class of functions to list is the entire set $\underline{n}^{\underline{k}}$. We can create a typical element in the list by choosing an element of $\underline{n}$ and writing it down, choosing another element (possibly the same as before) of $\underline{n}$ and writing it down next, and so on until we have made k decisions. This generates a function in one line form sequentially: First $f(1)$ is chosen, then $f(2)$ is chosen and so on. We can represent all possible decisions pictorially by writing down the decisions made so far and then some downward edges indicating the possible choices for the next decision.

The lefthand part of Figure 3.1 illustrates this way of generating a function in $\underline{2}^{\underline{3}}$ sequentially. It's called the *decision tree* for generating the functions in $\underline{2}^{\underline{3}}$. Each line in the left hand figure is labeled with the choice of function value to which it corresponds. Note that the labeling does not completely describe the corresponding decision—we should have used something like "Choose 1 for

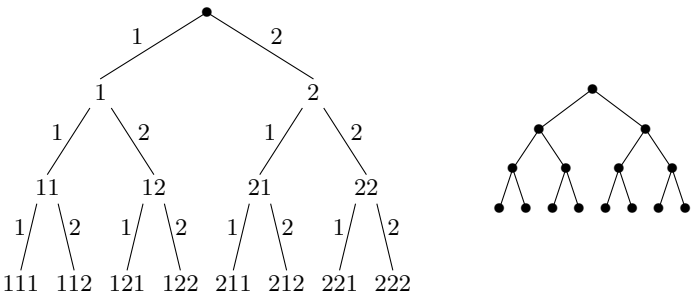


Figure 3.1 (a) The decision tree for all functions in $\underline{2}^{\underline{3}}$; that is, 3-long lists from $\{1, 2\}$ with repeats allowed is on the left. We've omitted commas in the lists. Functions are written in one line notation. (b) The underlying tree with all labels removed is on the right.

the value of $f(2)$" instead of simply "1" on the line from 1 to 11. At the end of each line is the function that has been built up so far. We've omitted commas, so 211 means 2,1,1.

To the right of the figure for generating the functions is the same structure without any labels. The dots (•) are called *nodes* or *vertices* and the lines connecting them are called *edges*. Sometimes it is more convenient to specify an edge by giving its two end points; for example, the edge from 1 to 12 in the figure can be described uniquely as (1,12). The nodes with no edges leading down are called the *leaves*. The entire branching structure is called a *tree* or, more properly, an *ordered rooted tree*. The topmost node is called the *root*.

In this terminology, the labels on the vertices show the partial function constructed so far when the vertex has been reached. The edges leading out of a vertex are labeled with all possible decisions that can be made next, given the partial function at the vertex. We labeled the edges so that the labels on edges out of each vertex are in order when read left to right. The leaves are labeled with the finished functions. Notice that the labels on the leaves are in lexicographic order. If we agree to label the edges from each vertex in order, then any set of functions generated sequentially by specifying $f(i)$ at the ith step will be in lex order.

To create a single function we start at the root and choose downward edges (i.e., make decisions) until we reach a leaf. This creates a *path* from the root to a leaf. We may describe a path in any of the following ways:

- the sequence of vertices $v_0, v_1, \ldots, v_m$ on the path from the root v_0 to the leaf v_m;
- the sequence of edges $e_1, e_2, \ldots, e_m$, where $e_i = (v_{i-1}, v_i)$, the edge connecting v_{i-1} to v_i;
- the integer sequence of *decisions* $D_1, D_2, \ldots, D_m$, where e_i has D_i edges to the left of it leading out from v_{i-1}.

Note that if a vertex has k edges leading out from it, the decisions are numbered 0, 1, ..., $k-1$. We will find the concept of a sequence of decisions useful in stating our algorithms in the next section. We illustrate the three approaches by looking at the leaf 2,1,2 in Figure 3.1:

- the vertex sequence is root, 2, 21, 212;
- the edge sequence is 2, 1, 2;
- the decision sequence is 1, 0, 1.

The following example uses a decision tree to list a set of patterns which are then used to solve a counting problem.

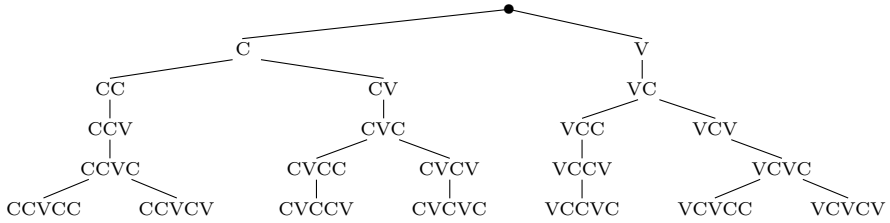


Figure 3.2 The decision tree for 5-long C-V patterns.

#V's	#CC's	ways to fill	patterns
1	2	$(20 \times 19)^2 \times 6$	CCVCC
2	0	$20^3 \times 6^2$	CVCVC
2	1	$(20 \times 19) \times 20 \times 6^2$	CCVCV CVCCV VCCVC VCVCC
3	0	$20^2 \times 6^3$	VCVCV

Figure 3.3 Grouping and counting the patterns.

Example 3.1 Counting words Using the 26 letters of the alphabet and considering the letters AEIOUY to be vowels how many five letter "words" (i.e. five long lists of letters) are there subject to the following constraints?

(a) No vowels are ever adjacent.

(b) There are never three consonants adjacent.

(c) Adjacent consonants are always different.

To start with, it would be useful to have a list of all the possible patterns of consonants and vowels; e.g., CCVCV (with C for consonant and V for vowel) is possible but CVVCV and CCCVC violate conditions (a) and (b) respectively and so are not possible. We'll use a decision tree to generate these patterns in lex order. Of course, a pattern CVCCV can be thought of as a function f where $f(1) = \text{C}$, $f(2) = \text{V}$, ..., $f(5) = \text{V}$. In Example 3.2 we'll use a decision tree for a counting problem in which there is not such a straightforward function interpretation.

We could simply try to list the patterns (functions) directly without using a decision tree. The decision tree approach is preferable because we are less likely to overlook something. The resulting tree is shown in Figure 3.2. At each vertex there are potentially two choices, but at some vertex only one is possible because of conditions (a) and (b) above. Thus there are one or two decisions at each vertex. You should verify that this tree lists all possibilities systematically.

* * * Stop and think about this! * * *

With enough experience, one can list the leaves of a fairly simple decision tree like this one without needing to draw the entire tree. It's easier to make mistakes using that short cut, so we don't recommend it—especially on exams!

We can now group the patterns according to how many vowels and how many adjacent consonants they have since these two pieces of information are enough to determine how many ways the pattern can be filled in. The results are given in Figure 3.3. To get the answer, we multiply the "ways to fill" in each row by the number of patterns in that row and sum over all four rows. The answer is $20^2 \times 6 \times 973$. □

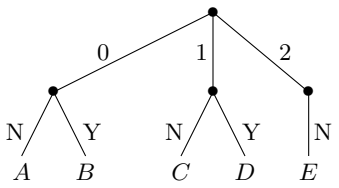


Figure 3.4 A decision tree for two 2 pair hands.

The next two examples also deal with patterns. The first example looks at patterns of overlap between two hands of cards and uses this to count hands. In contrast to the previous example, no locations are involved. The second example looks at another word problem. We take the patterns to be the number of repetitions of letters rather than the location of types of letters.

Example 3.2 Two hands of cards In Chapter 1 we studied various problems involving a hand of cards. Now we complicate matters by forming more than one hand on the same deal from the deck. How many ways can two 5 card hands be formed so that each hand contains 2 pairs (and a fifth card that has a different value)?

The problem can be solved by forming the first hand and then forming the second, since the number of choices for the second hand does not depend on what the first hand is as long as we know it is a hand with 2 pairs. We solved the two pair problem in Example 1.16 (p. 22), so we know that the first hand can be formed in 123,552 ways.

Forming the second hand is complicated by the fact that the first hand has used up some of the cards in the deck. As a result, we must consider different cases according to the amount of overlap between the first and second hands. We'll organize the possibilities by using a decision tree. Let P_i be the set of values of the pairs in the $i$thhand; e.g., we might have $P_1 = \{3, \mathrm{Q}\}$ and $P_2 = \{2, 3\}$, in which case the hands have one pair value in common. Our first decision will be the value of $|P_1 \cap P_2|$, which must be 0, 1 or 2. Our next decision will be based on whether or not the value of the unpaired card in the first hand is in P_2; i.e., whether or not a pair in the second hand has the same value as the nonpair card in the first hand. We'll label the edges Y and N according as this is true or not. The decision tree is shown in Figure 3.4, where we've labeled the leaves A–E for convenience.

We'll prove that the number of hands that correspond to each of the leaves is

$$
\begin{aligned}
A: &\quad \binom{10}{2}\binom{4}{2}^2(52-8-5) = 63{,}180,\\
B: &\quad \binom{3}{2} \times \binom{10}{1}\binom{4}{2}(52-8-4) = 7{,}200,\\
C: &\quad 2 \times \binom{10}{1}\binom{4}{2}(52-8-3) = 4{,}920,\\
D: &\quad 2\binom{3}{2}(52-8-2) = 252,\\
E: &\quad 52-8-1 = 43,
\end{aligned}
$$

giving a total of 75,595 choices for the second hand. Multiplying this by 123,552 (the number of ways of forming the first hand) we find that there are somewhat more than 9×10^9 possibilities for two hands. Of course, if the order of the hands is irrelevant, this must be divided by 2.

We'll derive the values for leaves A and D and let you do the rest. For A, we choose the two pairs not using any of the three values in the first hand. As in Example 1.16, this gives us $\binom{10}{2}\binom{4}{2}^2$ possibilities. We then choose the single card. To do the latter, we must avoid the values of the two pairs already chosen in the second hand and the cards in the first hand. Thus there are $52-8-5$ choices for this card. For D, we choose which element of P_1 is to be in P_2 (2 choices). The suits

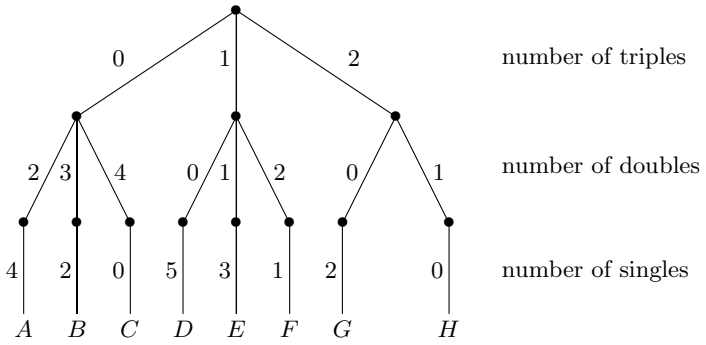


Figure 3.5 The decision tree for forming 8-letter words using ERRONEOUSNESS in Example 3.3. The first level of edges from the root indicates the number of different letters used three times; the next level, two times; and the bottom level, once.

of that pair in the second hand are determined since only two cards with that value are left in the deck after selecting the first hand. Next we choose the suits for the other pair ($\binom{3}{2}$ choices). Finally we select the fifth card. We must avoid (a) the values of the two pairs already chosen in the second hand and (b) the pair that was used in the first hand which doesn't have the same value as either of the pairs in the second hand. □

Example 3.3 Words from a collection of letters Review the problem discussed in Examples 1.11 (p. 13) and 1.19 (p. 24), where we counted possible "words" that could be made using the letters in ERROR. In Example 1.11, we counted by listing words one at a time. In Example 1.19, we counted by grouping words according to the number of occurrences of each letter. Both approaches could be carried out using a decision tree, for example, we could first choose m_1, then m_2 and finally m_3 in Example 1.19. Let's consider a bigger problem where it will be useful to form even larger groups of words than in Example 1.19. (Larger groups result in fewer total groups and so lead to easier counting.) We'll group words by patterns, as we did in Example 3.1, but the nature of the patterns will be different—it will be more like what we did for hands of cards in Chapter 1. Let's compute the number of 8-letter words that can be can be formed using the letters in ERRONEOUSNESS, using a letter no more often than it appears in ERRONEOUSNESS. We have three each of E and S; two each of O, N, and R; and one U. We will make three levels of decisions. The first level will be how many letters to use three times, the second level how many to use twice, and the third level how many to use once. Of course, since the total number of letters used must be 8 to create our 8-letter word, the choice on the third level is forced. Since there are only 6 distinct letters, we must either use at least one letter 3 times or use at least two letters twice. The decision tree is shown in Figure 3.5.

Why did we go from three times to twice to once in the levels instead of, for example in the reverse order?

* * * Stop and think about this! * * *

Had we first chosen how many letters were to appear once, the number of ways to choose letters to appear twice would depend on whether or not we had used U. The choices for triples would be even more complex. By starting with the most repeated letters and working down to the unrepeated ones, this problem does not arise. Had we made decisions in the reverse order just described, there would have been 23 leaves. You might like to try constructing the tree.

We can easily carry out the calculations for each of eight leaves. For example, at leaf F, we choose 1 of the 2 possibilities at the first choice AND 2 of the 4 possibilities at the second choice AND 1 of the 3 possibilities at the third choice. This gives $\binom{2}{1}\binom{4}{2}\binom{3}{1}$. Listing the choices at each

time in alphabetical order, they might have been E; N, S; O. AND now we choose locations for each of these letters. This can be counted by a multinomial coefficient: $\binom{8}{3,\,2,\,2,\,1}$. In a similar manner, we have the following results for the eight cases.

$$\begin{aligned}
A: &\quad \binom{2}{0}\binom{5}{2}\binom{4}{4}\binom{8}{2,\,2,\,1,\,1,\,1,\,1} = 100,800\\
B: &\quad \binom{2}{0}\binom{5}{3}\binom{3}{2}\binom{8}{2,\,2,\,2,\,1,\,1} = 151,200\\
C: &\quad \binom{2}{0}\binom{5}{4}\binom{2}{0}\binom{8}{2,\,2,\,2,\,2} = 12,600\\
D: &\quad \binom{2}{1}\binom{4}{0}\binom{5}{5}\binom{8}{3,\,1,\,1,\,1,\,1,\,1} = 13,440\\
E: &\quad \binom{2}{1}\binom{4}{1}\binom{4}{3}\binom{8}{3,\,2,\,1,\,1,\,1} = 107,520\\
F: &\quad \binom{2}{1}\binom{4}{2}\binom{3}{1}\binom{8}{3,\,2,\,2,\,1} = 60,480\\
G: &\quad \binom{2}{2}\binom{3}{0}\binom{4}{2}\binom{8}{3,\,3,\,1,\,1} = 6,720\\
H: &\quad \binom{2}{2}\binom{3}{1}\binom{3}{0}\binom{8}{3,\,3,\,2} = 1,680.
\end{aligned}$$

Adding these up, we get 454,440. This is about as simple as we can make the problem with the tools presently at our disposal. In Example 11.6 (p. 319), we'll see how to immediately identify the answer as the coefficient of a term in the product of some polynomials. □

In the previous examples we were interested in counting the number of objects of some sort—words, hands of cards. Decision trees can also be used to provide an orderly listing of objects. The listing is simply the leaves of the tree read from left to right. Figure 3.6 shows the decision tree for the permutations of $\underline{3}$ written in one line form. Recall that we can think of a permutation as a bijection $f : \underline{3} \to \underline{3}$ and its one line form is $f(1), f(2), f(3)$. Since we chose the values of $f(1)$, $f(2)$ and $f(3)$ in that order, the permuations are listed lexicographically; that is, in "alphabetical" order like a dictionary only with numbers instead of letters. On the right of Figure 3.6, we've abbreviated the decision tree a bit by shrinking the edges coming from vertices with only one decision and omitting labels on nonleaf vertices. As you can see, there is no "correct" way to label a decision tree. The intermediate labels are simply a tool to help you correctly list the desired objects (functions in this case) at the leaves. Sometimes one may even omit the function at the leaf and simply read it off the tree by looking at the labels on the edges or vertices associated with the decisions that lead from the root to the leaf. An example of such a tree appears in Figure 3.16 (p. 90).

Definition 3.1 Rank *The* **rank** *of a leaf is the number of leaves to its left. Thus, if there are n leaves the rank is an integer between 0 and $n-1$, inclusive.*

In Figure 3.6, the leaf 1,2,3 has rank 0 and the leaf 3,1,2 has rank 4. Rank is an important tool for storing information about objects and for generating objects at random. See the next section for more discussion.

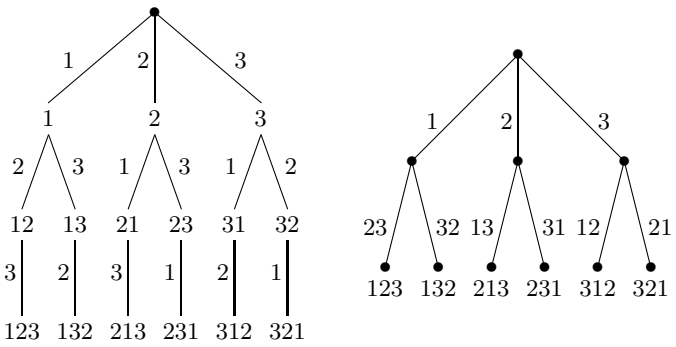


Figure 3.6 The decision tree for the permutations of $\underline{3}$ in lex order. (a) The full tree is on the left side. (b) An abbreviated tree is on the right. We've omitted commas and written permutations in one line form.

Example 3.4 Direct insertion order Another way to create a permutation is by *direct insertion*. (Often this is simply called "insertion.") Suppose that we have an ordered list of k items into which we want to insert a new item. It can be placed in any of $k+1$ places; namely, at the end of the list or immediately before the ith item where $1 \leq i \leq k$. By starting out with 1, choosing one of the two places to insert 2 in this list, choosing one of the three places to insert 3 in the new list and, finally, choosing one of the four places to insert 4 in the newest list, we will have produced a permutation of $\underline{4}$. To do this, we need to have some convention as to how the places for insertion are numbered when the list is written from left to right. The obvious choice is from left to right; however, right to left is often preferred because the leftmost leaf is then $12\ldots n$ as it is for lex order. We'll use the obvious choice (left to right) because it is less confusing.

Here's the derivation of the permutation associated with the insertions 1, 3 and 2.

number to insert	2	3	4	Answer
partial permutation	1	2 1	2 1 3	2 4 1 3
positions in list	1 2	1 2 3	1 2 3 4	
position to use	↑	↑	↑	

Figure 3.7 is the decision tree for generating permutations of $\underline{3}$ by direct insertion. The labels on the vertices are, of course, the partial permutations, with the full permutations appearing on the leaves. The decision labels on the edges are the positions in which to insert the next number. To the left of the tree we've indicated which number is to be inserted. This isn't really necessary since the numbers are always inserted in increasing order starting with 2. Notice that the labels on the leaves are no longer in lex order because we constructed the permutations differently. Had we labeled the vertices with the positions used for insertion, the leaves would then be labeled in lex order. If you don't see why this is so, label the vertices of the decision tree with the insertion positions.

Unlike lex order, it is not immediately clear that this method gives all permutations exactly once. We now prove this by induction on n. The case $n = 1$ is trivial since there are no insertions. Suppose $n > 1$ and that $a_1, \ldots, a_n$ is a permutation of $1, \ldots, n$. If $a_k = n$, then the last insertion must have been to put n in position k and the partial permutation of $1, \ldots, n-1$ before this insertion was $a_1, \ldots, a_{k-1}, a_{k+1}, \ldots, a_n$. By the induction assumption, the insertions leading to this permutation are unique and so we are done.

Like the method of lex order generation, the method of direct insertion generation can be used for other things besides permutations. However, direct insertion cannot be applied as widely as lex order. Lex order generation works with anything that can be thought of as an (ordered) list, but direct insertion requires more structure. □

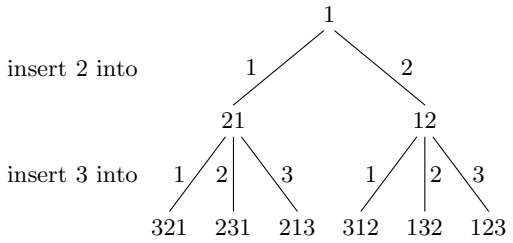


Figure 3.7 The decision tree for the permutations of $\underline{3}$ in direct insertion order. We've omitted commas.

Example 3.5 Transposition order Yet another way to create a permutation is by *transposition*. It is similar to direct insertion; however, instead of pushing a number into a space between elements of the partial permutation, we put it into a place that may already be occupied, bumping that element out of the way. If an element is bumped, it is moved to the end of the list.

Here's the derivation of the permutation associated with 1, 3 and 2. Note that, while direct insertion used positions between elements of the partial permutation, transposition uses the positions of the partial permutation plus one more space at the end.

number to insert	2	3	4	Answer
partial permutation	1	2 1	2 1 3	2 4 3 1
positions in list	1 2	1 2 3	1 2 3 4	
position to use	↑	↑	↑	

As with direct insertion, unique generation of all permutations is not obvious. A similar inductive proof works: Suppose the permutation is $a_1, \ldots, a_n$ and $a_k = n$. The preceding partial permutation is

- $a_1, \ldots, a_{n-1}$, if $k = n$,
- $a_1, \ldots, a_{k-1}, a_n, a_{k+1}, \ldots, a_{n-1}$, if $k < n$.

By the induction assumption, we can generate any $(n-1)$-long permutation, so generate whichever of the above permutations of $\{1, \ldots, n-1\}$ is required and then put n into position k, bumping if necessary. Thus the map from the $n!$ possible transposition sequences to permutations is a surjection. Since the domain and image of the map are the same size, the map is a bijection. This completes the proof.

Why is this called the transposition method? Suppose the sequence of positions is $p_2, \ldots, p_n$. Then the permutation is obtained by starting with the identity permutation and applying, in order, the transpositions $(2, p_2)$, $(3, p_3)$, $\ldots$, (n, p_n), where the pseudo-transposition (k, k) is interpreted as doing nothing.

Since this seems more complicated than lex order and insertion order, why would anyone use it? It is an easy way the generate random permutations: Generate a sequence $p_2, \ldots, p_n$ of random integers where $1 \le p_k \le k$. Then apply the method of the previous paragraph to create the random permutation. □

Example 3.6 Programs to list the leaves A decision tree consists of a set of sequential decisions, and so the code fragment discussed in connection with the Rule of Product is relevant. Here it is in decision tree language:

```
For each first decision d_1
   ...
      For each kth decision d_k
         List the structure arising from the decisions d_1, ..., d_k.
      End for
   ...
End for
```

We now give the code to generate the leaves of Figure 3.2. In `Create(a,b)`, we find the next possible decisions given the two preceding ones, `a` followed by `b`. Note that, if d_1 is the first decision, `Create(V,`d_1`)` generates the correct set of possible second decisions.

```
Function Create(a,b)                          /* Create next set of decisions. */
   If b = V
      Return {C}                              /* Avoid  VV pattern. */
   Elseif a = C
      Return {V}                              /* Avoid  CCC pattern. */
   Else
      Return {C, V}
   End if
End

Procedure ListCV
   D_1 = {C, V}
   For d_1 ∈ D_1:
      D_2 = Create( V,d_1)                    /*  V is fake zeroth decision. */
      For d_2 ∈ D_2:
         D_3 = Create(d_1,d_2)
         For d_3 ∈ D_3:
            D_4 = Create(d_2,d_3)
            For d_4 ∈ D_4:
               D_5 = Create(d_3,d_4)
               For d_5 ∈ D_5: List d_1d_2d_3d_4d_5 End for
            End for
         End for
      End for
   End for
End    □
```

Exercises

The following exercises are intended to give you some hands-on experience with simple decision trees before we begin our more systematic study of them in the next section.

3.1.1. What permutations of $\underline{3}$ have the same rank in lexicographic order and insertion order?

3.1.2. Draw the decision tree for the permutations of $\underline{4}$ in lexicographic order.

(a) What is the rank of 2,3,1,4? of 4,1,3,2?

(b) What permutation in this tree has rank 5? rank 15?

3.1.3. Draw the decision tree for the permutations of $\underline{4}$ in direct insertion order.

(a) What is the rank of 2314? of 4132?

(b) What permutation in this tree has rank 5? rank 15?

3.1.4. Draw the decision tree for the permutations of $\underline{4}$ in transposition order.

(a) What is the rank of 2314? of 4132?

(b) What permutation in this tree has rank 5? rank 15?

3.1.5. Draw a decision tree to list all 6-long sequences of A's and B's that satisfy the following conditions.

(i) There are no adjacent A's.

(ii) There are never three B's adjacent.

3.1.6. Draw the decision tree for the strictly decreasing functions in $\underline{6}^{\underline{4}}$ in lex order.

(a) What is the rank of 5,4,3,1? of 6,5,3,1?

(b) What function has rank 0? rank 7?

(c) What is the largest rank and which function has it?

(d) Your tree should contain the decision tree for the strictly decreasing functions in $\underline{5}^{\underline{4}}$. Indicate it and use it to list those functions in order.

(e) Indicate how all the parts of this exercise can be interpreted in terms of subsets of a set.

3.1.7. Draw the decision tree for the nonincreasing functions in $\underline{4}^{\underline{3}}$.

(a) What is the rank of 321? of 443?

(b) What function has rank 0? rank 15?

(c) What is the largest rank and which function has it?

(d) Your tree should contain the decision tree for the nonincreasing functions in $\underline{3}^{\underline{3}}$. Circle that tree and use its leaves to list the nonincreasing functions in $\underline{3}^{\underline{3}}$.

(e) Can you find the decision tree for the strictly decreasing functions in $\underline{4}^{\underline{3}}$ in your tree?

3.1.8. Describe the lex order decision tree for producing all the names in Example 2 in the Introduction to Part I.

*3.1.9. In a list of the permutations on $\underline{n}$ with $n > 1$, the permutation just past the middle of the list has rank $n!/2$.

(a) What is the permutation of $\underline{n}$ that has rank 0 in insertion order? rank $n!-1$? rank $n!/2$?

(b) What is the permutation of $\underline{n}$ that has rank 0 in transposition order? rank $n! - 1$? rank $n!/2$?

(c) What is the permutation of $\underline{n}$ that has rank 0 in lex order? rank $n! - 1$? rank $n!/2$ when n is even? rank $n!/2$ when n is odd?

Hint. For $n!/2$, look at some pictures of the trees.

3.1.10. How many ways can two full houses be formed?

3.1.11. How many ways can two 5-card hands be formed so that the first is a full house and the second contains two pair?

(a) Do this in the obvious manner by first choosing the full house and then choosing the second hand.

(b) Do this in the less obvious manner by first choosing the second hand and then choosing the full house.

(c) What lesson can be learned from the previous parts of this exercise?

3.1.12. How many ways can three full houses be formed?

3.1.13. How many 7-letter "words" can be formed from ERRONEOUSNESS where no letter is used more times than it appears in ERRONEOUSNESS?

*3.2 Ranking and Unranking

Suppose we have a decision tree whose leaves correspond to a certain class of objects; e.g., the permutations of $\underline{3}$. An important property of such a decision tree is that it leads to a bijection between the n leaves of the tree and the set $\{0, 1, \ldots, n-1\}$. In the previous section we called this bijection the rank of the leaf.

Definition 3.2 RANK and UNRANK *Suppose we are given a decision tree with n leaves. If v is a leaf, the function* RANK(v) *is the number of leaves to the left of it in the decision tree. Since RANK is a bijection from leaves to $\{0, 1, \ldots, n-1\}$, it has an inverse which we call UNRANK.*

How can these functions be used?

- Simpler storage: They can simplify the structure of an array required for storing data about the objects at the leaves. The RANK function is the index into the storage array. The UNRANK function tells which object has data stored at a particular array location. Thus, regardless of the nature of the N objects at the leaves, we need only have an N-long singly indexed array to store data.
- Less storage: Suppose we are looking at the 720 permutations of $\underline{6}$. Using RANK and UNRANK, we need a 720-long array to store data. If we took a naive approach, we might use a six dimensional array (one dimension for each of $f(1), \ldots, f(6)$), and each dimension would be able to assume six values. Not only is this more complicated than a singly dimensioned array, it requires $6^6 = 46,656$ storage locations instead of 720.
- Random generation: Suppose that you want to study properties of typical objects in a set of N objects, but that N is much too large to look at all of them. If you have a random number generator and the function UNRANK, then you can look at a random selection of objects: As often as desired, generate a random integer, say r, between 0 and $N-1$ inclusive and study the object UNRANK(r). This is sometimes done to collect statistics on the behavior of an algorithm.

You'll now learn a method for computing RANK and UNRANK functions for some types of decision trees. The basic idea behind all these methods is that, when you make a decision, all the decisions to the left of the one just made contribute the leaves of their "subtrees" to the rank of the leaf we are moving toward.

To talk about this conveniently, we need some terminology. Let $e = (v, w)$ be an edge of a tree associated with a decision D at vertex v; that is, e connects v to the vertex w that is reached by making decision D at v. The *residual tree of* (v, w), written $R(v, w)$ or $R(e)$, is v together with all edges and vertices that can be reached from v by starting with one of the D decisions to the left of (v, w). For example, the residual tree of the edge labeled 2 in Figure 3.6(b) consists of the edges labeled 1, 23 and 32 and the four vertices attached to these edges. The edge labeled 2 has $D = 1$. When $D = 0$, there are no edges to the left of e that have v as a parent. Thus, the *residual tree of* e consists of just one vertex when $D = 0$. Let $\Delta(e)$ be the number of leaves in $R(e)$, not counting the root. Thus $\Delta(e) = 0$ when e corresponds to $D = 0$ and $\Delta(e) > 0$ otherwise. The following result forms the basis of our calculations of RANK and UNRANK.

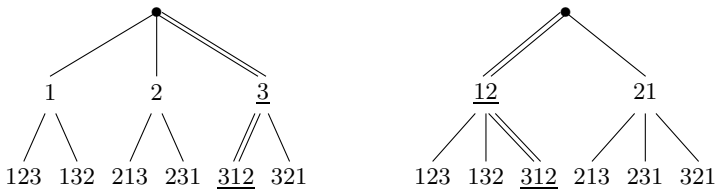


Figure 3.8 Decision trees for permutations of $\{1, 2, 3\}$. The decisions taken to reach 312 are shown in double lines. The lex order is on the left and direction insertion on the right. We've omitted commas.

Theorem 3.1 Rank formula *If the sequence of edges on the path from the root to a leaf X in a decision tree is e_1, e_2, ... e_m, then*

$$\text{RANK}(X) \;=\; \sum_{i=1}^{m} \Delta(e_i).$$

Before proving this let's look at some simple situations to see what's happening. In Figure 3.8 we have the lex and insertion order decision trees for permutations of $\underline{3}$ with the decisions needed to reach 312 drawn double. For lex order we have $\Delta(e) = 4$ for $e = (\bullet, 3)$ and $\Delta(e) = 0$ for $e = (3, 312)$. (We write this simply as $\Delta(\bullet, 3) = 4$ and $\Delta(3, 312) = 0$.) For the insertion order we have $\Delta(1, 12) = 0$ and $\Delta(12, 312) = 2$. This gives ranks of 4 and 2 in the two trees.

Proof: Now let's prove the theorem. We'll use induction on m. Let $e_1 = (v, w)$. The leaves to the left of X in the tree rooted at v are the leaves in $R(v, w)$ together with the leaves to the left of X in the tree starting at w. In other words, $\text{RANK}(X)$ is $\Delta(v, w)$ plus the rank of X in the tree starting at w.

Suppose that $m = 1$. Since $m = 1$, w must be a leaf and hence $w = X$. Thus the tree starting at w consists simply of X and so contains no leaves to the left of X. Hence X has rank 0 in that tree. This completes the proof for $m = 1$.

Suppose the theorem is true for decision sequences of length $m - 1$. Look at the subtree that is rooted at w. The sequence of edges from the root w to X is $e_2, \ldots, e_m$. Since $R(e_i)$ (for $i > 1$) is the same for the tree starting at w as it is for the full tree, it follows by the induction assumption that X has rank $\sum_{i=2}^{m} \Delta(e_i)$ in this tree. This completes the proof. □

Calculating RANK

Calculating $\text{RANK}(X)$ is, in principle, now straightforward since we need only obtain formulas for the $\Delta(e_i)$'s. Unfortunately, each decision tree is different and it is not always so easy to obtain such formulas. We'll work some examples to show you how it is done in some simple cases.

Example 3.7 Lex order rank of all functions If $D_1, \ldots, D_k$ is a sequence of decisions in the lex order tree for $\underline{n}^{\underline{k}}$ leading to a function $f \in \underline{n}^{\underline{k}}$, then $D_i = f(i) - 1$ since there are $f(i) - 1$ elements of $\underline{n}$ that are less than $f(i)$. What is the value of $\text{RANK}(f)$?

By Theorem 3.1, we need to look at $R(e_i)$, where e_i is an edge on the path. The structure of $R(e_i)$ is quite simple and symmetrical. There are D_i edges leading out from the root. Whichever edge we take, it leads to a vertex that has n edges leading out. This leads to another vertex with n edges leading out and so on until we reach a leaf. In other words, we make one D_i-way decision and $(k - i)$ n-way decisions. Each leaf of $R(e_i)$ is reached exactly once in this process and thus, by the Rule of Product $\Delta(e_i) = D_i n^{k-i}$. We have proved

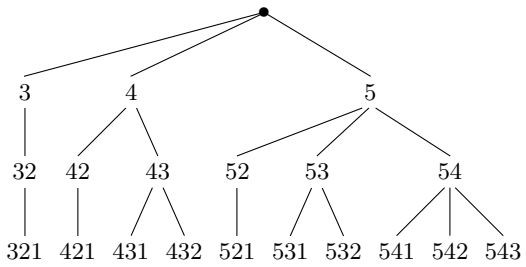


Figure 3.9 The lex order decision tree for the strictly decreasing functions in $\underline{5^3}$. We've omitted commas.

Theorem 3.2 Lex order rank of all functions *For f in the lex order listing of all functions in $\underline{n^k}$,*

$$\text{RANK}(f) \;=\; \sum_{i=1}^{k}(f(i)-1)n^{k-i}.$$

Notice that when $n = 10$ this is just a way of writing the number whose arabic notation is $D_1D_2\ldots D_k$. Indeed, the functions in $\underline{10^k}$ written in one line form look just like k digit numbers except that each digit has been increased by one. Lex order is then the same as numerical order.

This realization, generalized from base 10 to base n, gives another approach to the formula for RANK(f). It also makes it easy to see how to find the function that immediately follows f: Simply write out the set of decisions that lead to f as a number in base n and add 1 to this number to obtain the decisions for the function immediately following f. To get the function that is k further on, add k. To get the function that is k earlier, subtract k. (If we have to carry or borrow past the leftmost digit D_1, then we are attempting to go beyond the end of the tree.) For instance, the successor of 1,3,2,3,3 in $\underline{3^5}$ is obtained as follows. The decisions that lead to 1,3,2,3,3 are 0,2,1,2,2. Since there are three possible decisions at each point, we can think of the decisions as the base 3 number 02122 and add 1 to it to obtain 02200, which we think of as decisions. Translating from decisions to function values, we obtain 1,3,3,1,1 as the successor of 1,3,2,3,3. □

Example 3.8 Strictly decreasing functions We will now study the strictly decreasing functions in $\underline{n^k}$, which we have observed correspond to k element subsets of $\underline{n}$.

The decision tree for lex order with $n = 5$ and $k = 3$ is in Figure 3.9. You can use it to help you visualize some of the statements we'll be making.

Calculating the rank of a leaf differs from the preceding examples because the tree lacks symmetry; however, it has another feature which enables us to calculate the rank quite easily. Let's look for the rank of the strictly decreasing function $f \in \underline{n^k}$.

Let e_1 be the first edge on the path to the leaf for f. $R(e_1)$ is the tree for all strictly decreasing functions $g \in \underline{n^k}$ with $g(1) < f(1)$. In other words, $R(u, v)$ is the tree for the strictly decreasing functions in $\underline{f(1)-1^k}$.

We can generalize this observation. Suppose the path is $e_1,\ e_2, \ldots,\ e_k$. Suppose that g is a leaf of $R(e_i)$. This can happen if and only if $g \in \underline{n^k}$ is a strictly decreasing function with

$$g(1) = f(1),\ g(2) = f(2), \cdots, g(i-1) = f(i-1),\ \text{and } g(i) < f(i).$$

Since $g(j)$ is determined for $j < i$, look just at $g(i),\ g(i+1), \ldots,\ g(k)$. It is an arbitrary strictly decreasing function with initial value less than $f(i)$. Thus the leaves of $R(e_i)$ can be associated with the strictly decreasing functions in $\underline{f(i)-1^{\underline{k+1-i}}}$. Since there are $\binom{b}{a}$ strictly decreasing functions in $\underline{b^a}$, $\Delta(e_i) = \binom{f(i)-1}{k+1-i}$. Thus

Theorem 3.3 Lex order rank of strictly decreasing functions *For f in the lex order listing of all strictly decreasing functions in $\underline{n}^{\underline{k}}$,*

$$\text{RANK}(f) \;=\; \sum_{i=1}^{k} \binom{f(i)-1}{k+1-i}. \tag{3.1}$$

Let's use the theorem to calculate the rank of the strictly decreasing function $f = 96521$. What are k and n? Since there are five numbers in the list, we can see that $k = 5$; however, there's no way to determine n. This means that f has not been fully specified since we don't know its range. On the other hand, our formula doesn't require n and, as we remarked when defining functions, a specification of a function which omits the range may often be considered complete anyway. By our formula

$$\text{RANK}(96521) \;=\; \binom{8}{5} + \binom{5}{4} + \binom{4}{3} + \binom{1}{2} + \binom{0}{1} \;=\; 56+5+4+0+0 = 65.$$

What is the decision sequence for a strictly decreasing function? For the strictly decreasing functions in $\underline{n}^{\underline{k}}$, the decision associated with choosing $f(i) = j$ is $i + j - (k+1)$. Why is this so? We must choose distinct values for $f(i+1), \ldots, f(k)$. Thus there must be at least $k - i$ values less than j. Hence smallest possible value for $f(i)$ is $k + 1 - i$. Thus there are $j - (k + 1 - i)$ possible values less than j.

Note that (3.1) does not depend on n and so, for each k, there is one decision tree that works for all n. The root has an infinite number of children—one for each value of $f(1)$. Thereafter the number of decisions is finite since $f(i) < f(1)$ for $i > 1$. This "universal" decision tree arises because, once $f(1)$ is given, the value of n imposes no constraints on $f(i)$ for $i > 1$. Had we used strictly *increasing* functions instead, this would not have been the case because $n \geq f(i) > (1)$ for all $i > 1$. □

***Example 3.9 Direct insertion order rank of all permutations** Now let's compute ranks for the permutations of $\underline{n}$ in direct insertion order, a concept defined in Example 3.4. We can use practically the same approach that was used in Example 3.7. Number the positions where we can insert j as $0, 1, 2, \ldots, j$. (We started with 0, not 1 for the first position.) If $D_2,\ D_3,\ \ldots,\ D_n$ is the sequence of decisions, then D_j is the position into which j is to be inserted. Note that we've started the sequence subscripting with 2 so that the subscript j equals the number whose insertion is determined by D_j.

Since we usually tend to write permutations in one line form, we should develop a method for determining D_j from the one line form. Here is a simple rule:

> Write the permutation in one line form. Count the elements of the permutation *from right to left* up to but not including j and ignoring all elements that exceed j. That count is the position in which j was inserted and so equals D_j.

As an illustration, consider the permutation with the one line form 4,6,1,2,5,7,3. Starting from the right, counting until we reach 2 and ignoring numbers exceeding 2, we get that $D_2 = 0$. Similarly, $D_3 = 0$. Since we encounter 3, 2 and 1 before reaching 4, $D_4 = 3$. Only 3 intervenes when we search for 5, so $D_5 = 1$. Here's the full list:

$$D_2 = 0,\ D_3 = 0,\ D_4 = 3,\ D_5 = 1,\ D_6 = 4,\ D_7 = 1.$$

Again $R(e_i)$ has a nice symmetrical form as we choose a route to a leaf:

- Choose one of D_i edges AND
- Choose one of $i + 1$ edges (position for $i + 1$) AND

  . . . . . .
- Choose one of n edges (position for n).

By the Rule of Product, $\Delta(e_i) = D_i(i+1)(i+2)\cdots n \;=\; D_i n!/i!$ and so we have

Theorem 3.4 Direct insertion order rank *For f in the direct insertion order listing of all permutations on $\underline{n}$,*

$$\text{RANK}(f) \;=\; \sum_{i=2}^{n} \frac{D_i\, n!}{i!}.$$

This formula together with our previous calculations gives us

$$\begin{aligned}\text{RANK}(4,6,1,2,5,7,3) \;&=\; 0\times 7!/2! + 0\times 7!/3! + 3\times 7!/4! \\ &\quad + 1\times 7!/5! + 4\times 7!/6! + 1\times 7!/7! \;=\; 701.\end{aligned}$$

What permutation immediately follows $f = 9,8,7,3,1,6,5,2,4$ in direct insertion order? The obvious way to do this is to calculate $\text{UNRANK}(1+\text{RANK}(9,8,7,3,1,6,5,2,4))$, which is a lot of work. By extending an idea in the previous example, we can save a lot of effort. Here is a general rule:

> The next leaf in a decision tree is the one with the lexically next (legal) decision sequence.

Let's apply it. The decisions needed to produce f are 0,2,0,2,3,6,7,8. If we simply add 1 to the last decision, we obtain an illegal sequence because there are only nine possible decisions there. Just like in arithmetic, we reset it to 0 and carry. Repeating the process, we finally obtain 0,2,0,2,4,0,0,0. This corresponds to the sequence 1,3,1,3,5,1,1,1 of insertions. You should be able to carry out these insertions: start with the sequence 1, insert 2 in position 1, then insert 3 in position 3, and so on, finally inserting 9 in position 1. The resulting permutation is 3,6,1,5,2,4,7,8,9.

For special situations such as the ones we've been considering, one can find other short cuts that make some of the steps unnecessary. In fact, one need not even calculate the decision sequence. We will not discuss these short cuts since they aren't applicable in most situations. Nevertheless, you may enjoy the challenge of trying to find such shortcuts for permutations in lex and direct insertion order and for strictly decreasing functions in lex order. □

Calculating UNRANK

The basic principle for unranking is greed.

Definition 3.3 Greedy algorithm *A* **greedy algorithm** *is a multistep algorithm that obtains as much as possible at the present step with no concern for the future.*

The method of long division is an example of a greedy algorithm: If d is the divisor, then at each step you subtract off the largest possible number of the form $(k \times 10^n)d$ from the dividend that leaves a nonnegative number and has $1 \le k \le 9$.

The greedy algorithm for computing UNRANK is to choose D_1, then D_2 and so on, each D_i as large as possible at the time it is chosen. What do we mean by "as large as possible?" Suppose we are calculating UNRANK(m). If $D_1, \ldots, D_{i-1}$ have been chosen, then make D_i as big as possible subject to the condition that $\sum_{j=1}^{i} \Delta(e_j) \le m$.

Why does this work? Suppose that $D_1, \ldots, D_{i-1}$ have been chosen by the greedy algorithm and are part of the correct path. (This is certainly true when $i = 1$ because the sequence is empty!) We will prove that D_i chosen by the greedy algorithm is also part of the correct path.

Suppose that a path starts $D_1, \ldots, D_{i-1}, D'_i$. If $D'_i > D_i$, this cannot be part of the correct path because the definition of D_i gives $\Delta(e_1) + \cdots + \Delta(e_{i-1}) + \Delta(e'_i) > m$.

Now suppose that $D'_i < D_i$. Let x be the leftmost leaf reachable from the decision sequences that start $D_1, \ldots, D_i$. Clearly $\text{RANK}(x) = \Delta(e_1) + \cdots + \Delta(e_i) \le m$. Thus any leaf to the left of x will have rank less than m. Since all leaves reachable from D'_i are to the left of x, D'_i is not part of the correct decision sequence.

We have proven that if $D'_i \ne D_i$, then $D_1, \ldots, D_{i-1}, D'_i$ is not part of the correct path. It follows that $D_1, \ldots, D_{i-1}, D_i$ must be part of the correct path.

As we shall see, it's a straightforward matter to apply the greedy algorithm to unranking if we have the values of Δ available for various edges in the decision tree.

Example 3.10 Strictly decreasing functions

What strictly decreasing function f in $\underline{9}^{\underline{4}}$ has rank 100?

In view of Theorem 3.3 (p. 78), it is more natural to work with the function values than the decision sequence. Since $\Delta(e_i) = \binom{f(i)-1}{k+1-i}$, it will be handy to have a table of binomial coefficients to look at while calculating. Thus we've provided one in Figure 3.10. Since $k = 4$, the value of $f(1) - 1$ is the largest x such that $\binom{x}{4}$ does not exceed 100. We find $x = 8$ and $\binom{8}{4} = 70$. We now need to find x so that $\binom{x}{3} \le 100 - 70 = 30$. This gives 6 with $30 - 20 = 10$ leaves unaccounted for. With $\binom{5}{2} = 10$ all leaves are accounted for. Thus we get $\binom{0}{1}$ for the last Δ. Our sequence of values for $f(i) - 1$ is thus 8,6,5,0 and so $f = 9, 7, 6, 1$.

Although we specified that the domain of f was $\underline{9}$, the value 9 was never used in our calculations. This is like the situation we encountered when computing the rank. Thus, for ranking and unranking decreasing functions in $\underline{n}^{\underline{k}}$, there is no need to specify n.

Now let's find the strictly decreasing function of rank 65 when $k = 5$. The rank formula is

$$65 = \binom{f(1)-1}{5} + \binom{f(2)-1}{4} + \binom{f(3)-1}{3} + \binom{f(4)-1}{2} + \binom{f(5)-1}{1}. \qquad 3.2$$

Since $\binom{8}{5} = 56 < 65 < \binom{9}{5} = 126$, it follows that $f(1) = 9$. This leaves $65 - 56 = 9$ to account for. Since $\binom{5}{4} = 5 < 9 < \binom{6}{4} = 15$, we have $f(2) = 6$ and $9 - 5 = 4$ to account for. Since $\binom{4}{3} = 4$, $f(3) = 5$ and there is no more rank to account for. If we choose $f(4) \le 2$ and $f(5) \le 1$, the last two binomial coefficients in (3.2) will vanish. How do we know what values to choose? The key is to remember that f is *strictly* decreasing and takes on positive integer values. These conditions force $f(5) = 1$ and $f(4) = 2$. We got rid of the apparent multiple choice for f in this case, but will that always be possible?

* * * Stop and think about this! * * *

Yes, in any unranking situation there is at most one answer because each thing has a unique rank. Furthermore, if the desired rank is less than the total number of things, there will be some thing with that rank. Hence there is exactly one answer. □

	0	1	2	3
0	1			
1	1			
2	1	2		
3	1	3		
4	1	4	6	
5	1	5	10	
6	1	6	15	20
7	1	7	21	35

	0	1	2	3	4	5	6	7
8	1	8	28	56	70			
9	1	9	36	84	126			
10	1	10	45	120	210	252		
11	1	11	55	165	330	462		
12	1	12	66	220	495	792	924	
13	1	13	78	286	715	1287	1716	
14	1	14	91	304	1001	2002	3003	3432
15	1	15	105	455	1365	3003	5005	6435

Figure 3.10 Some binomial coefficients. The entry in row n and column k is $\binom{n}{k}$. For $n > k/2$ use $\binom{n}{k} = \binom{n}{n-k}$.

*Example 3.11 Direct insertion order What permutation f of $\underline{7}$ has rank 3,000 in insertion order?

Let the decision sequence be $D_2, \ldots, D_7$. The number of leaves in the residual tree for D_i is $D_i \times 7!/i!$ by our derivation of Theorem 3.4 (p. 79). Since $7!/2! = 2,520$, the greedy value for D_2 is 1. The number of leaves unaccounted for is $3,000 - 2,520 = 480$. Since $7!/3! = 840$, the greedy value for D_3 is 0 and 480 leaves are still unaccounted for. Since $7!/4! = 210$, we get $D_4 = 2$ and 60 remaining leaves. Using $7!/5! = 42$, $7!/6! = 7$ and $7!/7! = 1$, we get $D_5 = 1$, $D_6 = 2$ and $D_7 = 4$. Thus the sequence of decisions is 1,0,2,1,2,4, which is the same as insertion positions. You should be able to see that $f = 2, 4, 7, 1, 6, 5, 3$. □

*Gray Codes

Suppose we want to write a program that will have a loop that runs through all permutations of n. One way to do this is to run through numbers 0, ..., $n! - 1$ and apply UNRANK to each of them. This may not be the best way to construct such a loop. One reason is that computing UNRANK may be time consuming. Another, sometimes more important reason is that it may be much easier to deal with a permutation that does not differ very much from the previous one. For example, if we had n large blocks of data of various lengths that had to be in the order given by the permutation, it would be nice if we could produce the next permutation simply by swapping two adjacent blocks of data.

Methods that list the elements of a set so that adjacent elements in the list are, in some natural sense, close together are called *Gray codes*.

Suppose we are given a set of objects and a notion of closeness. How does finding a Gray code compare with finding a ranking and unranking algorithm? The manner in which the objects are defined often suggests a natural way of listing the objects, which leads to an efficient ranking algorithm (and hence a greedy unranking algorithm). In contrast, the notion of closeness seldom suggests a Gray code. Thus finding a Gray code is usually harder than finding a ranking algorithm. If we are able to find a Gray code, an even harder problem appears: Find, if possible, an efficient ranking algorithm for listing the objects in the order given by the Gray code.

All we'll do is discuss one of the simplest Gray codes.

Example 3.12 A Gray code for subsets of a set We want to look at all subsets of $\underline{n}$. It will be more convenient to work with the representation of subsets by n-strings of zeroes and ones: The string $s_1 \dots s_n$ corresponds to the subset S where $i \in S$ if and only if $s_i = 1$. Thus the all zeroes string corresponds to the empty set and the all ones string to $\underline{n}$.

Providing ranking and unranking algorithms is simple; just think of the strings as n-digit binary numbers. While adjacent strings in this ranking are close together numerically, their patterns of zeroes and ones may differ greatly. As we shall see, this tells us that the ranking doesn't give a Gray code.

Before we can begin to look for a Gray code, we must say what it means for two subsets (or, equivalently, two strings) to be close. Two strings will be considered close if they differ in exactly one position. In set terms, this means one of the sets can be obtained from the other by removing or adding a single element. With this notion of closeness, a Gray code for all subsets when $n = 1$ is $0,\ 1$. A Gray code for all subsets when $n = 2$ is $00,\ 01,\ 11,\ 10$.

How can we produce a Gray code for all subsets for arbitrary n? There is a simple recursive procedure. The following construction of the Gray code for $n = 3$ illustrates it.

0 00	1 10
0 01	1 11
0 11	1 01
0 10	1 00

You should read down the first column and then down the second. Notice that the sequences in the first column begin with 0 and those in the second with 1. The rest of the first column is simply the Gray code for $n = 2$ while the second column is the Gray code for $n = 2$, read from the last sequence to the first.

We now prove that this "two column" procedure for building a Gray code for subsets of an n-set from the Gray code for subsets of an $(n-1)$-set always works. Our proof will be by induction. For $n = 1$, we have already exhibited a Gray code. Suppose that $n > 1$ and that we have a Gray code for $n-1$. (This is the induction assumption.) We form the first column by listing the Gray code for $n-1$ and attaching a 0 at the front of each $(n-1)$-string. We form the second column by listing the Gray code, starting with the last and ending with the first, and attaching a 1 at the front of each $(n-1)$-string. Within a column, there is never any change in the first position and there is only a single change from line to line in the remaining positions because they are a Gray code by the induction assumption. Between the bottom of the first column and the top of the second, the only change is in the first position since the remaining $n-1$ positions are the last element of our Gray code for $n-1$. This completes the proof.

As an extra benefit, we note that the last element of our Gray code differs in only one position from the first element (Why?), so we can cycle around from the last element to the first by a single change.

It is a simple matter to draw a decision tree for this Gray code. In fact, Figure 3.1 is practically the $n = 3$ case—all we need to do is change some labels and keep track of whether we have reversed the code for the second column. (Reversing the code corresponds to interchanging the 0 and 1 edges.) The decision tree is shown in Figure 3.11. There is an easy way to decide whether the two edges leading down from a vertex v should be labeled 0-1 or 1-0: If the edge e leading into v is a 0 decision (i.e., 0 edge), use the same pattern that was used for e and the other decision e' it was paired with; otherwise, reverse the order. Another way you can think of this is that as you trace a path from the root to a leaf, going left and right, a 0 causes you to continue in the same direction and a 1 causes you to switch directions. □

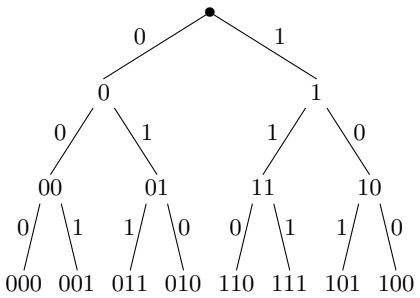


Figure 3.11 The decision tree for the Gray code for $\underline{3}$.

Exercises

Unless otherwise noted, the domain is $\underline{k}$ for some k and the range is $\underline{n}$ for some n. Usually, there is no need to specify n and specifying the function determines k.

3.2.1. This problem concerns strictly decreasing functions listed in lex order.

(a) Compute the rank of the functions 11,6,4 and 9,6,3,1. (Note that the domain of the first is $\underline{3}$ and the domain of the second is $\underline{4}$.)

(b) What strictly decreasing function with domain $\underline{4}$ has rank 35? rank 400?

(c) What strictly decreasing function immediately follows 9,6,3,2,1? 9,6,5,4?

(d) What strictly decreasing function immediately precedes 9,6,3,2,1? 9,6,5,4?

3.2.2. This problem concerns permutations listed in direct insertion order.

(a) Compute the ranks of 6,3,5,1,2,4 and 4,5,6,1,2,3.

(b) Determine the permutations of $\underline{6}$ with ranks 151 and 300.

(c) What permutation immediately follows 9,8,7,1,2,3,4,5,6? 6,5,4,1,2,3,7,8,9?

(d) What permutation immediately precedes 9,8,7,1,2,3,4,5,6? 6,5,4,1,2,3,7,8,9?

3.2.3. Consider the three strictly decreasing functions from $\underline{k}$ to $\underline{n}$:

(1) $k,\ k-1,\ldots,\ 2,\ 1$
(2) $k+1,\ k,\ldots,\ 3,\ 2$
(3) $k+2,\ k+1,\ldots,\ 4,\ 3$

Obtain simple formulas for their ranks.

3.2.4. This problem concerns nonincreasing functions listed in lex order.

(a) Prove that

$$\text{RANK}(f) \ = \ \sum_{i=1}^{k} \binom{f(i)+k-i-1}{k+1-i}.$$

Hint. Example 2.11 (p. 52) may be useful.

(b) Compute the ranks of 5,5,4,2,1,1 and 6,3,3.

(c) What nonincreasing function on $\underline{4}$ has rank 35? 400?

3.2.5. This problem concerns the permutations listed in lex order.

(a) Obtain a formula for RANK(f) in terms of the decisions $D_1,\ldots,D_n$ (or $D_1,\ldots,D_{n-1}$ if $f(n-1)$ and $f(n)$ are considered as being specified at the same time).

(b) Describe a method for going from a permutation in one line form to the decision sequence.

(c) Compute the ranks of 5,6,1,3,4,2 and 6,2,4,1,3,5.

(d) Compute the sequence of decisions for permutations of $\underline{6}$ which have ranks 151 and 300. What are the one line forms of the permutations?

3.2.6. Write computer code for procedures to calculate RANK and UNRANK. You might store the function being ranked as an array of integers where the first component is the size of the domain. The classes of functions for which you should write procedures are:

(a) the strictly decreasing functions in $\underline{n}^{\underline{k}}$ in lex order;

(b) the nonincreasing functions in $\underline{n}^{\underline{k}}$ in lex order.

(c) the permutations of $\underline{n}$ in insertion order;

(d) the permutations of $\underline{n}$ in lex order;

Hint. For the first two, it might be useful to have a procedure for calculating the binomial coefficients $C(a, b)$.

3.2.7. Returning to Example 3.12, draw the decision tree for $n = 4$ and list the Gray code for $n = 5$.

3.2.8. Returning to Example 3.12, compute the ranks of the following sequences in the Gray code for subsets of $\underline{n}$:

$$0110; \quad 1001; \quad 10101; \quad 1000010; \quad 01010101.$$

(The value of n is just the length of the sequence.)

3.2.9. Returning to Example 3.12 for each of the following values of n and k, find the n-string of rank k in the Gray code for subsets of $\underline{n}$:

$$n = 20,\ k = 0; \quad n = 20,\ k = 2^{19}; \quad n = 4,\ k = 7; \quad n = 8,\ k = 200.$$

*3.2.10. We will write the permutations of $\underline{n}$ in one line form. Two permutations will be considered adjacent if one can be obtained from the other by interchanging the elements in two adjacent positions. We want a Gray code for all permutations. Here is such a code for $n = 4$. You should read down the first column, then the second and finally the third.

1,2,3,4	3,1,2,4	2,3,1,4
1,2,4,3	3,1,4,2	2,3,4,1
1,4,2,3	3,4,1,2	2,4,3,1
4,1,2,3	4,3,1,2	4,2,3,1
4,1,3,2	4,3,2,1	4,2,1,3
1,4,3,2	3,4,2,1	2,4,1,3
1,3,4,2	3,2,4,1	2,1,4,3
1,3,2,4	3,2,1,4	2,1,3,4

List a Gray code for $n = 5$. As a challenge, describe a method for listing a Gray code for general n.

*3.3 Backtracking

In many computer algorithms it is necessary to systematically inspect all the vertices of a decision tree. A procedure that systematically inspects all the vertices is called a *traversal of the tree.* How can we create such a procedure? One way to imagine doing this is to walk around the tree. An example is shown in Figure 9.2 (p. 249), where we study the subject in more depth. "Walking around the tree" is not a very good program description. We can describe our traversal more precisely by giving an algorithm. Here is one which traverses a tree whose leaves are associated with functions and lists the functions in the order of their rank.

Theorem 3.5 Systematic traversal algorithm *The following procedure systematically visits the leaves in a tree from left to right by "walking" around the tree.*

1. Start: *Mark all edges as unused and position yourself at the root.*
2. Leaf: *If you are at a leaf, list the function.*
3. Decide case: *If there are no unused edges leading out from the vertex, go to Step 4; otherwise, go to Step 5.*
4. Backtrack: *If you are at the root, STOP; otherwise, return to the vertex just above this one and go to Step 3.*
5. Decision: *Select the leftmost unused edge out of this vertex, mark it used, follow it to a new vertex and go to Step 2.*

If you cannot easily visualize the entire route followed by this algorithm, take the time now to apply this algorithm to the decision tree for $\underline{2}^{\underline{3}}$ shown in Figure 3.1 and verify that it produces all eight functions in lex order.

Because Step 1 refers to all leaves, the algorithm may be impractical in its present form. This can be overcome by keeping a "decision sequence" which is updated as we move through the tree. Here's the modified algorithm.

Theorem 3.6 Systematic traversal algorithm (programming version) *The following procedure systematically visits the leaves in a tree from left to right by "walking" around the tree.*

1. Start: *Initialize the decision sequence to -1 and position yourself at the root.*
2. Leaf: *If you are at a leaf, list the function.*
3. Decide case: *Increase the last entry in the decision sequence by 1. If the new value equals the number of decisions that can be made at the present vertex, go to Step 4; otherwise go to Step 5.*
4. Backtrack: *Remove the last entry from the decision sequence. If the decision sequence is empty, STOP; otherwise go to Step 3.*
5. Decision: *Make the decision indicated by the last entry in the decision sequence, append -1 to the decision sequence and go to Step 2.*

In both versions of the algorithm, Step 4 is labeled "Backtrack." What does this mean? If you move your pencil around a tree, this step would correspond to going toward the root on an edge that has already been traversed in the opposite direction. In other words, backtracking refers to the process of moving along an edge *back* toward the root of the tree. Thinking in terms of the decision sequence, backtracking corresponds to undoing (i.e., backtracking on) what is currently the last decision made.

If we understand the structure of the decision tree well enough, we can avoid parts of the algorithm In the next example, we eliminate the descent since only one element in the Gray code changes each time. In the example after that, we eliminate the decision sequence for strictly decreasing functions. In both cases, we use `empty` to keep track of the state of the decision sequence. Also, Step 2 is avoided until we know we are at a leaf.

Example 3.13 Listing Gray coded subsets In Example 3.12 we looked at a Gray code for listing all elements of an n element set. Since there are only two decisions at each vertex, the entries in the decision sequence will be 0 or 1 (but they usually *do not* equal the entries in the Gray code). Here is the code with s_i being the decision sequence and g_i the Gray code.

```
Procedure GraySubsets(n)
   For i = 1 to n:                          /* Set up first leaf */
      s_i = 0
      g_i = 0
   End for
   empty = FALSE
   While (empty=FALSE):
      Output g_1, ..., g_n.                 /* Step 2 */
      For i from n to 1 by -1:              /* Steps 3 & 4 (moving up) */
         If (s_i = 0) then
            s_i = 1
            g_i = 1 - g_i                   /* Step 5 (moving right) */
            For j from i + 1 to n:
               s_j = n + 1 - j              /* Steps 3 & 5 (moving down) */
            End for
            Goto ENDCASE
         End if
         empty = TRUE
         Label ENDCASE
      End for
   End while
End
```

The statement $g_i = 1 - g_i$ changes 0 to 1 and vice versa. Since the Gray code changes only one entry, we are able to move down without changing any of the g_j values.

You may find this easier to understand than the presentation in Example 3.12. In that case, why don't we just throw that example out of the text? Although the code may be easy to follow, there is no reason to believe that it lists all subsets. Example 3.12 contains the proof that the algorithm does list all subsets, and the proof given there requires the recursive construction of the gray code based on reversing the order of the output for subsets of an $n-1$ element set. Once we know that it works, we do the backtracking using the fact that only one g_i is changed at each output. Thus, we can forget about the construction of the Gray code as soon as we know that it works and a little about the decision tree. □

Example 3.14 Listing strictly decreasing functions In Example 3.8 we studied ranking and unranking for strictly decreasing functions from $\underline{k}$ to $\underline{n}$, which correspond to the k element subsets of $\underline{n}$. Now we want to list the subsets by traversing the decision tree.

Suppose we are at the subset given by $d_1 > \cdots > d_k$. Recall that, since the d_i are strictly decreasing and $d_k \geq 1$, we must have $d_i \geq k+1-i$. We must back up from the leaf until we find a vertex that has unvisited children. If this is the $(i-1)$st vertex in the list, its next child after d_i will be $d_i - 1$ and the condition just mentioned requires that $d_i - 1 \geq k+1-i$. Here is the code for the traversal

```
Procedure Subsets(n, k)
   For i from 1 to k:  d_i = n + 1 - i  End for
   empty = FALSE
   While (empty=FALSE):
      Output d_1, ..., d_k.                    /* Step 2 */
      For i from k to 1 by -1:                 /* Steps 3 & 4 (moving up) */
         If (d_i > k + 1 - i) then
            d_i = d_i - 1                      /* Step 5 (moving right) */
            For j from i + 1 to k:
               d_j = n + 1 - j                 /* Steps 3 & 5 (moving down) */
            End for
            Goto ENDCASE
         End if
         empty = TRUE
         Label ENDCASE
      End for
   End while
End    □
```

So far in our use of decision trees, it has always been clear what decisions are reasonable; i.e., lead, after further decisions, to a solution of our problem. This is because we've looked only at simple problems such as generating *all* permutations of $\underline{n}$ or generating *all* strictly decreasing functions in $\underline{n}^{\underline{k}}$. Consider the following problem.

$$\begin{aligned} &\text{How many permutations } f \text{ of } \underline{n} \text{ are such that} \\ &|f(i) - f(i+1)| \leq 3 \quad \text{for} \quad 1 \leq i < n? \end{aligned} \tag{3.3}$$

It's not at all obvious what decisions are reasonable in this case. For instance, when $n = 9$, the partially specified one line function 124586 cannot be completed to a permutation.

There is a simple cure for this problem: We will allow ourselves to make decisions which lead to "dead ends," situations where we cannot continue on to a solution. With this expanded notion of a decision tree, there are often many possible decision trees that appear reasonable for doing something. We'll look at this a bit in a minute. For now, let's look at our problem (3.3). Suppose that we're generating things in lex order and we've reached the vertex 12458. What do we do now? We'll simply continue to generate more of the permutation, making sure that (3.3) is satisfied for that portion of the permutation we have generated so far. The resulting portion of the tree that starts at 1,2,4,5,8 is shown in Figure 3.12. Each vertex is labeled with the part of the permutation after 12458. The circled leaves are solutions.

Our tree traversal algorithm given at the start of this section requires a slight modification to cover our extended decision tree concept where a leaf need not be a solution: Change Step 2 to

2′. Solution?: If you are at a solution, take appropriate action.

How can there be more than one decision tree for generating solutions in a specified order? Suppose someone who was not very clever wanted to generate all permutations of $\underline{n}$ in lex order.

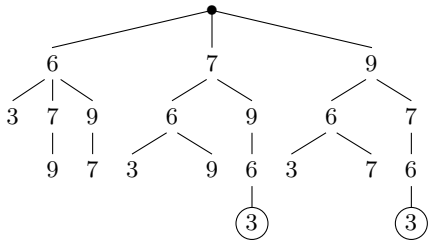


Figure 3.12 The portion of the decision tree for (3.3) that begins 1,2,4,5,8. The decision that led to a vertex is placed at the vertex rather than on the edge. The circled leaves are solutions.

He might program a computer to generate all functions in $\underline{n}^{\underline{n}}$ in lex order and to then discard those functions which are not permutations. This leads to a much bigger tree because n^n is much bigger than $n!$, even when n is as small as 3. A somewhat cleverer friend might suggest that he have the program check to see that $f(k) \neq f(k-1)$ for each $k > 1$. This won't slow down the program very much and will lead to only $n(n-1)^{n-1}$ functions. Thus the program should run faster. Someone else might suggest that the programmer check at each step to see that the function produced so far is an injection. If this is done, nothing but permutations will be produced, but the program may be much slower. Someone more knowledgeable might suggest a way to convert a decision sequence to a permutation using the ideas of the previous section. This would give the smallest possible tree and would not take very long to run. You might try to figure out how to do that.

The lesson to be learned from the previous paragraph is that there is often a trade off between the size of the decision tree and the time that must be spent at each vertex determining what decisions to allow. (For example, the decision to allow only those values for $f(k)$ which satisfy $f(k) \neq f(k-1)$.) Because of this, different people may develop different decision trees for the same problem. The differences between computer run times for different decision trees can be truly enormous. By carefully limiting the decisions, people have changed problems that were too long to run on a supercomputer into problems that could be easily run on a personal computer.

We'll conclude this section with two examples of backtracking of the type just discussed.

Example 3.15 Latin squares

An $n \times n$ *Latin square* is an $n \times n$ array in which each element of $\underline{n}$ appears exactly once in each row and column. Let $L(n)$ be the number of $n \times n$ Latin Squares. Finding a simple formula, or even a good estimate, for $L(n)$ is an unsolved problem. How can we use backtracking to compute $L(n)$ for small values of n?

The number of Latin Squares increases rapidly with n, so anything we can do to reduce the size of the decision tree will be a help. Here's a way to cut our work by a factor of $n!$. Let's agree to rearrange the columns of a Latin Square so that the first row always reads $1, 2, 3, \ldots, n$. We'll say such a square is "first row ordered." Given a first row ordered square, we can permute the columns in $n!$ ways, each of which leads to a different Latin Square. Hence $L(n)$ is $n!$ times the number of first row ordered Latin Squares.

By next rearranging the rows, we can get the entries in the first column in order, too. If we're sloppy, we might think this gives us another factor of $n!$. This is not true because 1 is already at the top of the first column due to our ordering of the first row. Hence only the second through nth positions are arbitrary and so we have a factor of $(n-1)!$.

Let's organize what we've got now. We'll call a Latin Square in *standard form* if the entries in the first row are in order and the entries in the first column are in order. Each $n \times n$ Latin Square in standard form is associated with $n!\ (n-1)!$ Latin Squares which can be obtained from it by permuting all rows but the first and then permuting all columns. The standard form Latin Squares

1	1 2	1 2 3	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4
	2 1	2 3 1	2 1 4 3	2 1 4 3	2 3 4 1	2 4 1 3
		3 1 2	3 4 1 2	3 4 2 1	3 4 1 2	3 1 4 2
			4 3 2 1	4 3 1 2	4 1 2 3	4 3 2 1

Figure 3.13 The standard Latin squares for $n \leq 4$.

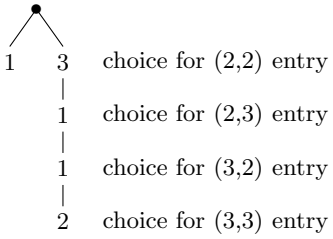


Figure 3.14 A decision tree for 3×3 Latin squares.

for $n \leq 4$ are shown in Figure 3.13. It follows that $L(1) = 1 \times 1!\ 0! = 1$, $L(2) = 1 \times 2!\ 1! = 2$, $L(3) = 1 \times 3!\ 2! = 12$ and $L(4) = 4 \times 4!\ 3! = 576$.

We'll set up the decision tree for generating Latin Squares in standard form. Fill in the entries in the array in the order (2,2), (2,3), ..., (2,n), (3,1),..., (3,n), ..., (n, n); that is, in the order we read—left to right and top to bottom. The decision tree for $n = 3$ is shown in Figure 3.14, where each vertex is labeled with the decision that leads to it. The leaf on the bottom right corresponds to the standard form Latin Square for $n = 3$. The leaf on the left is the end of a path that has died because there is no way to choose entry (2,3). Why is this? At this point, the second row contains 1 and 2, and the third column contains 3. The (2,3) entry must be different from all of these, which is impossible. □

Example 3.16 Arranging dominoes A backtracking problem is stated in Example 4 at the beginning of this part: We asked for the number of ways to systematically arrange 32 dominoes on a chessboard so that each one covers exactly two squares and every square is covered.

If the squares of the board are numbered systematically from 1 to 64, we can describe any placement of dominoes by a sequence of 32 h's and v's: Place dominoes sequentially as follows. If the first unused element in the sequence is h, place a horizontal domino on the first unoccupied square and the square to its right. If the first unused element in the sequence is v, place a vertical domino on the first unoccupied square and the square just below it. Not all sequences correspond to legal arrangements because some lead to overlaps or to dominoes off the board. For a 2×2 board, the only legal sequences are hh and vv For a 2×3 board, the legal sequences are hvh, vhh and vvv. For a 3×4 board, there are eleven legal sequences as shown in Figure 3.15.

To find these sequences in lex order we used a decision tree for generating sequences of h's and v's in lex order. Each decision is required to lead to a domino that lies entirely on the board and does not overlap another domino. The tree is shown in Figure 3.16. Each vertex is labeled with the choice that led to the vertex. The leaf associated with the path $vhvvv$ does not correspond to a covering. It has been abandoned because there is no way to place a domino on the lower left square of the board, which is the first free square. Draw a picture of the board to see what is happening.

Our systematic traversal algorithm can be used to traverse the decision tree without actually drawing it; however, there is a slight difficulty: It is not immediately apparent what the possible decisions at a given vertex are. This is typical in backtracking problems. (We slid over it in the

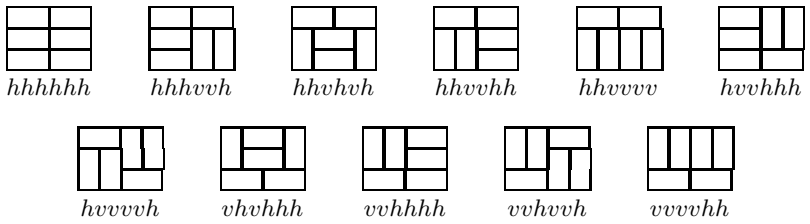


Figure 3.15 The domino coverings of a 3×4 board in lex order.

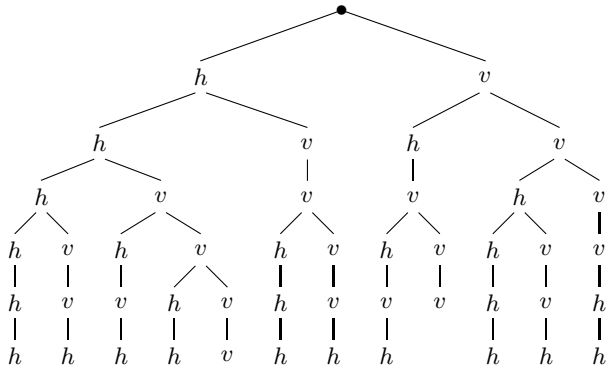


Figure 3.16 The lex order decision tree for domino coverings of a 3×4 board. The decision that led to a vertex is placed at the vertex rather than on the edge.

previous example.) A common method for handling the problem is to keep track of the nature of the decision (h or v) rather than its numerical value. For example, suppose our decision sequence is $v, h, -1$ and we are at Step 3 (Decide case). We attempt to increase -1 to h and find that this is impossible because it results in a domino extending off the board. Thus we try to increase it to v, which is acceptable. After a few more moves through the tree, we arrive Step 3 with the sequence $v, h, v, v, v, -1$. When we try to increase -1 to h, the domino overlaps another. When we try v, the domino extends off the board. Hence the vertex is a leaf but not a solution, so we go to Step 4. □

Exercises

3.3.1. Draw a decision tree for the 4×4 standard form Latin Squares.

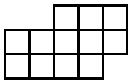
3.3.2. The "n queens problem" is to place n queens on an $n \times n$ chessboard so that no queen attacks another. For those not familiar with chess, one queen attacks another if it is in the same row, column or diagonal. Convince yourself that the 2 queens and 3 queens problems have no solutions and that the 4 queens problem has two solutions. A necessary, *but not sufficient*, condition for a solution is that each row contain a queen. Thus we can imagine a sequence of n decisions to obtain a solution: The ith decision is where to place a queen in the ith row.

(a) Find the 2 solutions to the 4 queens problem by drawing a decision tree.

(b) Draw a decision tree for the 5 queens problem. How many solutions are there?

(c) How many solutions are there to the 6 queens problem?

3.3.3. Draw a decision tree for covering the following board with dominoes.



3.3.4. Draw a decision tree for covering a 3×3 board using the domino and the L-shaped 3×3 square pattern: .

Notes and References

Although listing is a natural adjunct to enumeration, it has not received as much attention as enumeration until recently. Listing, ranking and unranking have emerged as important topics in connection with the development of algorithms in computer science. These are now active research areas in combinatorics.

The text [1] discusses decision trees from a more elementary viewpoint and includes applications to conditional probability calculations. The text by Stanton and White [3] is at the level of this chapter. Williamson [5; Chs.1,3] follows the same approach as we do, but explores the subject more deeply. A different point of view is taken by Nijenhuis and Wilf [2, 4].

1. Edward A. Bender and S. Gill Williamson, *Mathematics for Algorithm and Systems Analysis*, Dover (2005).
2. Albert Nijenhuis and Herbert S. Wilf, *Combinatorial Algorithms*, Academic Press (1975).
3. Dennis Stanton and Dennis White, *Constructive Combinatorics*, Springer-Verlag (1986).
4. Herbert S. Wilf, *Combinatorial Algorithms: An Update*, SIAM (1989).
5. S. Gill Williamson, *Combinatorics for Computer Science*, Dover (2002).

CHAPTER 4

Sieving Methods

Introduction

A "sieving method" is a technique that allows us to count or list some things indirectly. After a few words about organization and difficulty, we'll introduce the two sieving methods discussed in this chapter.

- The sections of this chapter are independent of each other. Thus, if your instructor assigns only the material on the Principle of Inclusion and Exclusion, you need not read the sections on structures with symmetries. You may also read the material on counting structures with symmetries without reading the material on listing them.
- The material in this chapter is more difficult than the first three chapters in this part. Since the material here is not needed until Part IV, it may be postponed.

Structures Lacking Things

In Section 4.1 we look at the problem of counting structures that lack certain things; e.g., lists with no repeated elements or permutations with no fixed points. Sometimes, as in the case of lists with no repeated elements, it is easy to count the structures directly. That situation is not of interest here. Instead, we'll examine what happens when it's fairly easy to count structures which have some of the properties but hard to count those which have none of the properties. For example, consider permutations of $\underline{n}$ and a set $\{F_1, \ldots, F_n\}$ of n properties where F_i is the property that the permutation fixes i; that is, maps i to i. Suppose our problem is to count permutations with none of the properties; that is, permutations with no fixed points. This is hard. However, it is fairly easy to count permutations whose fixed points include some specified set S; that is, permutations that have at least some the properties $\{F_i \mid i \in S\}$. These counts can be used to indirectly solve the original problem by using the "Principle of Inclusion and Exclusion."

The Principle of Inclusion and Exclusion can be extended in various ways. We briefly indicate some of these at the end of Section 4.1.

Structures with Symmetries

At the end of Example 1.12 (p. 13), we asked how many ways we could form a six long circular sequence using ones and twos and found that we could not solve it. In this section we'll develop the necessary tools.

The circular sequence problem is difficult because "symmetries induce equivalences." What does this mean? The sequence 121212 looks the same if it is circularly shifted two positions. This is a symmetry of the sequence. Several sequences correspond to the same circular sequence of ones and twos. We say these lists are "equivalent." Thus, as we saw in Example 1.12, the three sequences 112112, 121121 and 211211 are equivalent. We can find a sequence equivalent to a given one by reading the given sequence "circularly:" Start reading at any point. At the end of the sequence jump to the start and continue until you return to where you began reading.

To list the circular sequences, we need a list C of sequences such that every sequence is equivalent to exactly one sequence in C. Thus, exactly one of the sequences 112112, 121121 and 211211 would appear in C. Counting the circular sequences means finding $|C|$. We'll discuss listing first and then counting.

We have already dealt with one important case of symmetries, namely, when our structures are lists and we are allowed to permute the items in the list in any fashion whatsoever. In other words, two lists are the same if they can be made identical by permuting the elements in one of them. In fact, this case is so important that it has a name: multisets. (Remember that a multiset is simply a list where order is irrelevant.) If the elements of the multiset can be ordered, then we can take our representatives C to be a collection of nondecreasing functions. This was discussed in Section 2.3.

In Section 4.2 we'll look at the problem of listing structures when symmetries are present. This is much like the nonmathematical notion of a sieve: all that comes through the sieve are "canonical" representations of the structures. Decision trees play an important role.

In Section 4.3 we'll look at the problem of counting, rather than listing, these structures. "Burnside's Lemma" provides us with an indirect method for doing this.

4.1 The Principle of Inclusion and Exclusion

Imagine that a professor on the first day of class wants to obtain information on the course background of the students. He wants to know what number of students have had Math 21, what number have had Comp Sci 13 and various combinations such as "Comp Sci 13 but not Math 21." For some reason, to calculate these numbers the professor asks just the following three questions.

"How many of you have had Math 21?"
"How many of you have had Comp Sci 13?"
"How many of you have had Comp Sci 13 and Math 21?"

Suppose the number of students is 15, 12 and 8, respectively.

Can the professor now determine answers to all other possible questions concerning having taken or not taken these courses? Let's look at a couple of possibilities.

How many have had Comp Sci 13 but not Math 21? Of the 12 students who have had the first course, 8 have had the second and so $12 - 8 = 4$ of them have not had the second.

How many students have had neither course? That will depend on the total number of students in the class. Suppose there are 30 students in the class. We might think that $30 - 15 - 12 = 3$ of them have had neither course. This is not correct because the students who had both courses were subtracted off twice. To get the answer, we must add them back in once. The result is that there are $3 + 8 = 11$ students who have had neither course.

We can rephrase the previous discussion in terms of sets. Let S be the set of students in the class, S_1 the subset who have had Math 21 and S_2 the subset who have had Comp Sci 13. The information that was obtained by questioning the class can be written as

$$|S| = 30, \quad |S_1| = 15, \quad |S_2| = 12 \quad \text{and} \quad |S_1 \cap S_2| = 8,$$

where $S_1 \cap S_2$ denotes the intersection of the sets S_1 and S_2. We saw that the number of students who had neither course is given by

$$|S| - \Big(|S_1| + |S_2|\Big) + |S_1 \cap S_2|. \tag{4.1}$$

How can this result be extended to more than two classes? The answer is provided by the following theorem. After stating it, we'll see how it can be applied before proving it.

Theorem 4.1 Principle of Inclusion and Exclusion *Let $S_1, S_2, \ldots, S_m$ be subsets of a set S. Let $N_0 = |S|$ and, for $r > 0$, let*

$$N_r \;=\; \sum |S_{i_1} \cap \cdots \cap S_{i_r}|, \tag{4.2}$$

where the sum is over all r-long strictly increasing sequences chosen from $\underline{m}$; that is, $\{i_1, \ldots, i_r\}$ ranges over all r-subsets of $\underline{m}$. The number of elements in S that are not in any of $S_1, \ldots, S_m$ is

$$\sum_{i=0}^{m} (-1)^i N_i \;=\; N_0 - N_1 + N_2 \cdots + (-1)^m N_m. \tag{4.3}$$

When $m = 2$, the one long sequences are 1 and 2, giving $N_1 = |S_1| + |S_2|$. The only two long sequence is 1,2 and so $N_2 = |S_1 \cap S_2|$. Thus (4.3) reduces to (4.1) in this case.

As we saw in an earlier chapter, strictly increasing sequences are equivalent to subsets. Also, the order in which we do intersections of sets does not matter. (Just as the order of addition does not matter and the order of multiplication does not matter.) This explains why we could have said that the sum defining N_r was over all r-subsets $\{i_1, \ldots, i_r\}$ of $\underline{m}$.

One can rewrite (4.3) in a somewhat different form. To begin with, the Rule of Sum tells us that $|S|$ equals the number of things not in any of the S_i's plus the number of things that are in at least one of the S_i's. The latter equals

$$|S_1 \cup \cdots \cup S_m|.$$

Using (4.3) and noting that $N_0 = |S|$, we have

$$N_0 \;=\; |S_1 \cup \cdots \cup S_m| + N_0 - N_1 + N_2 \cdots + (-1)^m N_m.$$

Rearranging leads to

Corollary *With the same notation as in Theorem 4.1 (p. 95),*

$$|S_1 \cup \cdots \cup S_m| \;=\; \sum_{i=1}^{m} (-1)^{i-1} N_i. \tag{4.4}$$

In this form, the Principle of Inclusion and Exclusion can be viewed as an extension of the Rule of Sum: The Rule of Sum tells us that if $T = S_1 \cup \cdots \cup S_m$ and if each structure in T appears in exactly one of the S_i, then

$$|T| \;=\; |S_1| + |S_2| + \ldots + |S_m|.$$

The left hand side of this equation is the left hand side of (4.4). The right hand side of this equation is N_1, the first term on the right hand side of (4.4). The remaining terms on the right hand side of (4.4) can be thought of as "corrections" to the Rule of Sum due to the fact that elements of T can appear in more than one S_i.

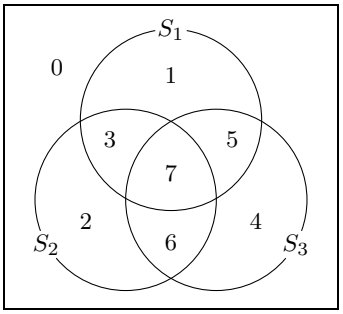


Figure 4.1 A Venn diagram for three subsets S_1, S_2 and S_3.

Example 4.1 Venn diagrams When m is quite small in Theorem 4.1, it is possible to draw a picture, called a *Venn diagram* that illustrates the theorem. Figure 4.1 shows such a diagram for $m = 3$. The interior of the box should be thought of as containing points which are the elements of S. (These points are not actually shown in the diagram.) Similarly, the interior of the circle labeled S_1 contains the elements of S_1 and its exterior contains the points not in S_1. Altogether, the three circles for S_1, S_2 and S_3 divide the box into eight regions which we have numbered 0 through 7.

In the figure, region 7 corresponds to $S_1 \cap S_2 \cap S_3$. Region 0 corresponds to those elements of S that are not in any S_i. Region 3 corresponds to those elements of S that are in S_1 and S_2 but not in S_3. You should be able to describe all of the other regions in a similar manner. The elements of S_1 are those in the four regions numbered 1, 3, 5 and 7. The elements of $S_1 \cap S_3$ are those in regions 5 and 7. You should be able to describe all the intersections in the Principle of Inclusion and Exclusion in this manner. You can then determine how often each region is counted in N_r and thereby obtain a proof of (4.3) for $m = 3$. It is possible to generalize this argument to prove (4.3), but we will give a slightly different proof of (4.3) later. □

Example 4.2 Using the theorem Many of Alice's 16 friends are athletic—they cycle, jog or swim on a regular basis. In fact we know that 6 of them cycle, 6 of them jog, 6 of them swim, 4 of them cycle and jog, 2 of them cycle and swim, 3 of them jog and swim and 2 of them engage in all three activities. How many of Alice's friends do none of these things on a regular basis?

Let S be the set of all friends, S_1 the set that cycle, S_2 the set that jog and S_3 the set that swim. We will apply (4.3) with $m = 3$. The information we were given can be rewritten as follows:

$$\begin{aligned}
N_0 &= 16 \quad \text{since} \quad |S| = 16 \\
N_1 &= 18 \quad \text{since} \quad |S_1| = 6 \quad |S_2| = 6 \quad |S_3| = 6; \\
N_2 &= 9 \quad \text{since} \quad |S_1 \cap S_2| = 4 \quad |S_1 \cap S_3| = 2 \quad |S_2 \cap S_3| = 3; \\
N_3 &= 2 \quad \text{since} \quad |S_1 \cap S_2 \cap S_3| = 2.
\end{aligned}$$

Thus the answer to our question is that $16 - 18 + 9 - 2 = 5$ of her friends neither cycle nor jog nor swim regularly. □

At this point you may well object that this method is worse than useless because there are much easier ways to get the answer. For example, to find out how many students took neither Math 21 nor Comp Sci 13, it would be easier to simply ask "How many of you have had neither Math 21 nor Comp Sci 13?" This is true. So far we've just been getting familiar with what the Principle of Inclusion and Exclusion means. We now turn to some examples where it is useful.

Example 4.3 Counting surjections How many surjections are there from $\underline{n}$ to $\underline{k}$?

This problem is closely related to $S(n,k)$, the Stirling numbers of the second kind, which we studied previously but couldn't find a formula for. In fact, a surjection f defines a partition of the domain into k blocks where the ith block is $f^{-1}(i)$. Since the blocks are all distinct and $S(n,k)$ does not care about the order of the blocks, the number of surjections is $k!S(n,k)$. Our attention will be devoted to the surjections—we don't need $S(n,k)$ here—but we pointed out the connection because it will allow us to get a formula for $S(n,k)$, too.

Let S be the set of all functions from $\underline{n}$ to $\underline{k}$ and let S_i be the set of those functions that never take on the value i. In this notation, the set of surjections is the subset of S that does not belong to any of $S_1, \ldots, S_k$ because a surjection takes on all values in its range. This suggests that we use (4.3) with $m = k$.

We found long ago that $|S| = k^n$. It is equally easy to find $|S_{i_1} \cap \cdots \cap S_{i_r}|$: The set whose cardinality we are taking is just all functions from $\underline{n}$ to $\underline{k} - \{i_1, \ldots, i_r\}$ and so equals $(k-r)^n$. This tells us that each of the terms in the sum (4.2) defining N_r equals $(k-r)^n$. Consequently, N_r is $(k-r)^n$ times the number of terms. Since there $\binom{k}{r}$ subsets of $\underline{k}$ of size r, the sum contains $\binom{k}{r}$ terms and $N_r = \binom{k}{r}(k-r)^n$. It follows from (4.3) that the number of surjections is

$$\sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n. \tag{4.5}$$

Combining this with the remarks at the start of this example, we have

$$S(n,k) \;=\; \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n \;=\; \sum_{i=0}^{k} \frac{(-1)^i}{i!} \frac{(k-i)^n}{(k-i)!}. \tag{4.6}$$

Because of the possibility of considerable cancellation due to alternating signs, numerical evaluation of this expression for large values of n and k can be awkward. □

In learning to apply the Principle of Inclusion and Exclusion, it can be difficult to decide what the sets $S, S_1, \ldots$ should be. It is often helpful to think in terms of

- a larger problem S that is easier to solve and
- conditions C_i that *all* do NOT hold for precisely those structures in S that are solutions of the original problem.

What's the connection between all this and the sets in Theorem 4.1? The set S_i is the set of structures in the larger problem S that satisfy C_i. Note that NOT appears in our description because (4.3) counts those elements of S that are NOT in any of the S_i's.

Let's look at the previous example in these terms. Our larger problem is the set of all functions for $\underline{n}$ to $\underline{k}$; that is, $S = \underline{k}^{\underline{n}}$. Since we want those functions that do NOT omit any of the values $1, \ldots, k$, we take C_i to be the condition that $f : \underline{n} \to \underline{k}$ omits the value i; that is, $i \notin \text{Image}(f)$.

Sometimes people talk about properties instead of conditions. In this case, they speak of "having a property" instead of "satisfying a condition."

Example 4.4 Counting solutions to equations How many different solutions are there to the equation

$$x_1 + x_2 + x_3 + x_4 + x_5 = n, \tag{4.7}$$

where the x_i's must be positive integers, none of which exceeds k?

Since it is easier to solve the equations without the constraint that the x_i's not exceed k, we'll use Theorem 4.1 as follows:

- Let S be the set of all positive integer solutions $(x_1, \ldots, x_5)$ of the equation.
- Let the ith condition be $x_i > k$ for $i = 1$, 2, 3, 4 and 5. The answer to the original problem is the number of elements of S (i.e., positive integer solutions) that satisfy none of the conditions.

For this to be useful, we must be able to easily determine, for example, how many solutions to (4.7) have $x_1 > k$ and $x_3 > k$. This is simply the number of solutions to $y_1 + \cdots + y_5 = n - 2k$ because we can take $x_1 = y_1 + k$, $x_3 = y_3 + k$ and $x_j = y_j$ for $j = 2$, 4 and 5.

We are ready to apply (4.3) with $m = 5$. To begin with, what is $|S|$? A solution in S can be obtained by inserting commas and plus signs in the $n - 1$ spaces between the n ones in $(1\ 1\ 1 \ldots 1)$ in such a way that either a plus or a comma, but not both, is inserted in each space and exactly 4 commas are used. Thus $|S| = \binom{n-1}{4}$. By this and the end of the previous paragraph, it follows that

$$|S_{i_1} \cap \ldots \cap S_{i_r}| \;=\; \binom{n - kr - 1}{4}, \tag{4.8}$$

where the binomial coefficient is taken to be zero if $n - kr - 1 < 4$. Since there are $\binom{5}{r}$ choices for the set $\{i_1, \ldots, i_r\}$, the number of solutions to (4.7) is

$$\begin{aligned}\binom{n-1}{4} &- \binom{5}{1}\binom{n-k-1}{4} + \binom{5}{2}\binom{n-2k-1}{4} \\ &- \binom{5}{3}\binom{n-3k-1}{4} + \binom{5}{4}\binom{n-4k-1}{4} \\ &- \binom{5}{5}\binom{n-5k-1}{4}.\end{aligned}$$

This formula is a bit tricky. If we blindly replace the binomial coefficients using the falling factorial formula

$$\binom{m}{4} \;=\; \frac{m(m-1)(m-2)(m-3)}{24} \tag{4.9}$$

and use algebra to simplify the result, we will discover that the number of solutions is zero! How can this be? The definition of binomial coefficient that we used for (4.8) gives $\binom{m}{4} = 0$ when $m < 0$, which does not agree with the falling factorial formula (4.9). Thus (4.9) cannot be used when $m < 0$.

The problem that we have been considering can be interpreted in other ways:

- How many compositions of n are there that consist of five parts, none of which exceed k?
- How many ways can n unlabeled balls be placed into five labeled boxes so that no box has more than k balls?

You should easily be able to see that these problems are all equivalent. □

Finally, the proof of Theorem 4.1:

Proof: Suppose that $s \in S$. Let $X \subseteq \underline{k}$ be such that $x \in X$ if and only if $s \in S_x$; that is, X is the set consisting of the indices of those S_i's that contain s. How much does s contribute to the sum in (4.3)? For the theorem to be true, it must contribute 1 if $X = \emptyset$ and 0 otherwise.

Clearly s contributes 1 to the sum when $X = \emptyset$, but what happens when $X \neq \emptyset$?

To begin with, what does s contribute to N_r when $r > 0$? It contributes nothing to some terms and contributes 1 to those terms in N_r for which $s \in S_{i_1} \cap \cdots \cap S_{i_r}$. By the definition of X, this happens if and only if $\{i_1, \ldots, i_r\} \subseteq X$ Thus s contributes to precisely those terms of N_r that correspond to subsets of X. Since there are $\binom{|X|}{r}$ such terms, s contributes $\binom{|X|}{r}$ to N_r. This is 0 if $r > |X|$. Thus the contribution of s to (4.3) is

$$1 + \sum_{r=1}^{k} (-1)^r \binom{|X|}{r} \;=\; \sum_{r=0}^{|X|} \binom{|X|}{r} (-1)^r.$$

By the binomial theorem, this sum is $(1-1)^{|X|} = 0^{|X|}$, which is zero when $|X| > 0$.

A different proof using "characteristic functions" is given in Exercise 4.1.10. □

Example 4.5 Derangements Recall that a derangement of $\underline{n}$ is a permutation f such that $f(x) = x$ has no solutions; i.e., the permutation has no cycles of length 1. A cycle of length 1 is also called a "fixed point." Let D_n be the number of derangements of $\underline{n}$. What is the value of D_n?

Let the set S of objects be all permutations of $\underline{n}$ and, for $1 \leq i \leq n$, let S_i be those permutations having i as a fixed point. In other words, the larger problem is counting all permutations. If σ is a permutation, condition C_i states that $\sigma(i) = i$.

The set $S_{i_1} \cap \cdots \cap S_{i_r}$ consists of those permutations for which the r elements of $I = \{i_1, \ldots, i_r\}$ are fixed points. Since such permutations can be thought of as permutations of $\underline{n} - I$, there are $(n-r)!$ of them. Thus $N_r = \binom{n}{r}(n-r)! = n!/r!$. By (4.3),

$$D_n \;=\; n! \sum_{i=0}^{n} \frac{(-1)^i}{i!}. \tag{4.10}$$

We will use the following theorem from calculus to obtain a simple approximation to D_n.

Theorem 4.2 Alternating series *Suppose that $|b_0| \geq |b_1| \geq |b_2| \geq \ldots$, that the values of b_k alternate in sign and that $\lim_{k\to\infty} b_k = 0$. Then $\sum_{k=0}^{\infty} b_k$ converges and*

$$\left| \sum_{k=0}^{\infty} b_k - \sum_{k=0}^{n} b_k \right| \;\leq\; |b_{n+1}|.$$

The terms in (4.10) alternate in sign and decrease in magnitude. Since

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{k!} \;=\; \frac{1}{e},$$

it follows that D_n differs from $n!/e$ by at most $\frac{n!}{(n+1)!} = \frac{1}{n+1}$. Hence, for $n > 1$, D_n is the closest integer to $n!/e$. □

Exercises

4.1.1. Let us define a "typical four letter word" to be a string of four letters $L_1L_2L_3L_4$ where L_1 and L_4 are consonants and at least one of L_2 and L_3 is a vowel.

(a) For $i = 2$ and $i = 3$, let V_i be the set of sequences $L_1L_2L_3L_4$ where L_1 and L_4 are consonants, L_i is a vowel and the remaining letter is arbitrary. Draw a Venn diagram for the two sets V_2 and V_3. Indicate what part of the diagram corresponds to typical four letter words and calculate the number of such words by using the Rule of Product and the Principle of Inclusion and Exclusion.

(b) For $i = 2$ and $i = 3$, let C_i be the set of sequences $L_1L_2L_3L_4$ where L_1, L_i and L_4 are consonants and the remaining letter is arbitrary. Draw a Venn diagram for the two sets C_2 and C_3. Indicate what part of the diagram corresponds to typical four letter words and calculate the number of such words.

4.1.2. How many ways can n married couples be paired up to form n couples so that each couple consists of a man and a woman and so that no couple is one of the original married couples?

4.1.3. Charled Dodgson (Lewis Carroll) Speaks of a battle among 100 combatants in which 80 lost an arm, 85 a leg, 70 an eye and 75 an ear. (Yes, it's gruesome, but that's the way he stated it.) Some number p of people lost all four.

(a) It is possible that p could be as large as 70? Why?

*(b) Find a lower bound for p and explain how your lower bound could actually be achieved.
Hint. A key to getting a lower bound is to realize that there are only 100 people.

4.1.4. How many ways can we make an n-card hand that contains at least one card from each each of the 4 suits?
Hint. Let a property of a hand be the absence of a suit.

4.1.5. Let $\varphi(N)$ be the number of integers between 1 and N inclusive that have no factors in common with N. Thus $\varphi(1) = \varphi(2) = 1$, $\varphi(3) = \varphi(4) = \varphi(6) = 2$ and $\varphi(5) = 4$. φ is called the *Euler phi function*. Let $p_1, \ldots, p_n$ be the primes that divide N. For example, when $N = 300$, the list of primes is $2, 3, 5$. Let S_j be the set of $x \in \underline{N}$ such that x is a divisible by p_j, or, equivalently, the jth property is that p_j divides the number.

(a) Prove that (4.3) determines $\varphi(N)$.

(b) Prove

$$|S_{i_1} \cap \cdots \cap S_{i_r}| = \frac{N}{p_{i_1} \cdots p_{i_r}}.$$

(c) Use this to prove $\varphi(N) = N \prod_{k=1}^{n} \left(1 - \frac{1}{p_k}\right)$.

4.1.6. Call an $n \times n$ matrix A of zeroes and ones *bad* if there is an index k such that $a_{k,i} = a_{i,k} = 0$ for $1 \le i \le n$. In other words, the row and column passing through (k, k) consist entirely of zeroes. Let $g(n)$ be the number of $n \times n$ matrices of zeroes and ones which are *not* bad.

(a) For any subset K of $\underline{n}$, let $z(K)$ be the number of $n \times n$ matrices A of zeroes and ones such that $a_{i,j} = 0$ if either i or j or both belong to K. Explain why $z(K)$ depends only on $|K|$ and obtain a simple formula for $z(K)$. Call it z_k where $k = |K|$.

(b) Express $g(n)$ as a fairly simple sum in terms of z_k.

4.1.7. Let C be the multiset $\{c_1, c_1, c_2, c_2, \ldots, c_m, c_m\}$ containing two copies each of m distinct symbols. How many ways can the elements of S be arranged in an ordered list so that adjacent symbols are distinct.
Hint. A list in which c_i and c_i are adjacent can be thought of as a list made from the multiset

$$C \cup \{c_i^2\} - \{c_i, c_i\},$$

where c_i^2 is a new symbol that stands for c_ic_i in the list.

*4.1.8. This is the same as the previous exercise, except that now each of c_1 through c_m appears in C three times instead of twice. The constraint is still the same: Adjacent symbols must be distinct.
Hint. There are now two types of properties, namely c_ic_i appearing in the list and $c_ic_ic_i$ appearing in the list. Call the corresponding sets S_i^2 and S_i^3. In computing N_r you need to consider how many times you require an S_i^3 and how many times you require an S_j^2 as well as S_j^3.

4.1.9. Let $(S, \Pr)$ be a probability space and let $S_1, \ldots, S_m$ be subsets of S. Prove that

$$\Pr((S_1 \cup \cdots \cup S_m)^c) \;=\; \sum_{i=0}^{m} (-1)^i N_i \quad \text{where} \quad N_r \;=\; \sum \Pr(S_{i_1} \cap \cdots \cap S_{i_r}),$$

the sum ranging over all r-long strictly increasing sequences chosen from $\underline{m}$.

4.1.10. The goal of this exercise is to use "characteristic functions" to prove Theorem 4.1 (p. 95). Let $\chi_i : S \to \{0, 1\}$ be the characteristic function of S_i; that is,

$$\chi_i(s) \;=\; \begin{cases} 1 & \text{if } s \in \S_i \\ 0 & \text{if } s \notin S_i. \end{cases}$$

(a) Explain why the number we want in the theorem is $\displaystyle\sum_{s \in S} \prod_{i=1}^{m} \bigl(1 - \chi_i(s)\bigr)$.

(b) Prove that

$$\prod_{i=1}^{m} \bigl(1 - \chi_i(s)\bigr) \;=\; \sum_{I \subseteq \underline{m}} (-1)^{|I|} \prod_{i \in I} \chi_i(s).$$

(c) Complete the proof of Theorem 4.1.

4.1.11. We want to count the number of elements in exactly k of the S_i. Let K^c be the complement of K relative to $\underline{m}$; that is, $K^c = \underline{m} \backslash K$.

(a) Explain why the number we want is

$$\sum_{s \in S} \sum_{\substack{K \subseteq \underline{m} \\ |K| = k}} \Biggl(\prod_{i \in K} \chi_i(s) \Biggr) \Biggl(\prod_{i \in K^c} \bigl(1 - \chi_i(s)\bigr) \Biggr).$$

(b) Show that this expression is

$$\sum_{s \in S} \sum_{\substack{K \subseteq \underline{m} \\ |K| = k}} \sum_{J \subseteq J^c} (-1)^{|J|} \prod_{i \in J \cup K} \chi_i(s).$$

(c) Show that this equals

$$\sum_{s \in S} \sum_{L \subseteq \underline{m}} \sum_{\substack{K \subseteq L \\ |K| = k}} (-1)^{|L|-k} \Biggl(\prod_{i \in L} \chi_i(s) \Biggr) \;=\; \sum_{s \in S} \sum_{L \subseteq \underline{m}} (-1)^{|L|-k} \Biggl(\prod_{i \in L} \chi_i(s) \Biggr) \binom{|L|}{k}.$$

(d) Conclude that the number of elements in S that belong to exactly k of the S_i is

$$\sum_{\ell=k}^{m} (-1)^{\ell-k} \binom{\ell}{k} N_\ell.$$

*Bonferroni's Inequalities

We conclude this section by looking briefly at two more advanced topics related to the Principle of Inclusion and Exclusion: Bonferroni's Inequalities and partially ordered sets.

Theroem 4.1 can sometimes be a bit of a problem to use even after we've formulated our problem and know exactly what we must count. There are two reasons for this. First, there will be a lot of addition and subtraction to do if m is large. Second, it may be difficult to actually compute values of N_r so we may have to be content with estimating them for small values of r and ignoring them when r is large. Because of these problems, we may prefer to obtain a quick approximation to (4.3). A method that is frequently useful for doing this is provided by the following theorem.

Theorem 4.3 Bonferroni's inequalities *Let the notation be the same as in Theorem 4.1 (p. 95) and let E be the number of elements of S not in any of the S_i. Then*

$$-N_t \;\le\; E - \sum_{r=0}^{t-1}(-1)^r N_r \;\le\; N_t;$$

i.e., truncating the sum gives an error which is no larger than the first term that was neglected. Furthermore, the sum is either an overestimate or an underestimate according as t is odd or even, respectively.

We can't prove this simply by appealing to Theorem 4.2 (p. 99) because the terms may be increasing in size. The proof of Bonferroni's Inequalities is left as an exercise.

Example 4.6 Using the theorem Let $r(n,k)$ be the fraction of those functions in $\underline{k^n}$ which are surjections. Using Bonferroni's inequalities and the ideas in Example 4.3 (p. 97), we'll estimate $r(n,k)$.

Let's begin with $t = 2$ in the theorem. In that case we simply need to divide the $i = 0$ and $i = 1$ terms in (4.5) by the total number of functions. Thus

$$r(n,k) \;\ge\; \frac{k^n - k(k-1)^n}{k^n} \;=\; 1 - k(1-1/k)^n.$$

With $k = 10$ and $n = 40$, we see that at least 85.2% of the functions in $\underline{10^{40}}$ are surjections.

If we set $t = 3$ in Bonferroni's inequalities, we obtain the upper bound

$$r(n,k) \le \frac{k^n - k(k-1)^n + \binom{k}{2}(k-2)^n}{k^n} \;=\; 1 - k(1-1/k)^n + \binom{k}{2}(1-2/k)^n.$$

With $k = 10$ and $n = 40$, we see that at most 85.8% of the functions in $\underline{10^{40}}$ are surjections. □

*Partially Ordered Sets

There is an important generalization of the Principle of Inclusion and Exclusion (Theorem 4.1 (p. 95)) which we'll just touch on. It requires some new concepts.

A *binary relation* ρ on a set S is a subset of $S \times S$. Instead of writing $(x, y) \in \rho$, people write $x\rho y$. For example, if S is a set of integers, then we can let ρ be the set of all $x, y \in S$ with x less than y. Thus, $x\rho y$ if and only if x is less than y. People usually use the notation $<$ for this binary relation. As another example, we can let S be the set of all subsets of $\underline{n}$ and let $\subseteq$ be the binary relation.

We can describe equivalence relations as binary relations: Let $\sim$ be an equivalence relation on S. Those pairs (x, y) for which $x \sim y$ form a subset of $S \times S$.

We now define another important binary relation.

Definition 4.1 Partially Ordered Set *A set P and a binary relation ρ satisfying*

(P-1) *$x\rho x$ for all $x \in P$;*

(P-2) *if $x\rho y$ and $y\rho x$, then $x = y$; and*

(P-3) *if $x\rho y$ and $y\rho z$, then $x\rho z$*

is called a **partially ordered set**, *also called a poset. The binary relation ρ is called a* **partial order**.

The real numbers with $x\rho y$ meaning "x is less than or equal to y" is a poset. The subsets of a set with $x\rho y$ meaning $x \subseteq y$ is a poset. Because of these examples, people often use the symbol $\leq$ or the symbol $\subseteq$ in place of ρ, even when the partial order does not involve numbers or subsets.

We now return to Theorem 4.1 and begin by rewriting the terms in (4.2) as functions of sets: $f(\{i_1, \ldots, i_r\}) = |S_{i_1} \cap \cdots \cap S_{i_r}|$. How should we define $f(\emptyset)$? It should be the size of the empty intersection. In most situations, the best choice for the empty intersection is everything. Thus we should probably take $f(\emptyset) = |S|$, the size of the set that contains everything.

Many people find this choice for the empty intersection confusing, so we digress briefly to explain it. (If it does not confuse you, skip to the next paragraph.) Let's look at something a bit more familiar—summations. As you know, the value of

$$g(A) \;=\; \sum_{i \in A} h(i) \tag{4.11}$$

is defined to be the sum of $f(i)$ over all $i \in A$. You should easily see that if A and B are disjoint nonempty sets, then

$$g(A \cup B) \;=\; g(A) + g(B). \tag{4.12}$$

If we want this to be true for $B = \emptyset$, we must have

$$g(A) + g(\emptyset) \;=\; g(A \cup \emptyset) \;=\; g(A), \tag{4.13}$$

and so $g(\emptyset)$ must equal 0, the identity element for addition. Suppose we replace the sum in (4.11) with a product. Then (4.12) becomes $g(A \cup B) = g(A)g(B)$ and the parallel to (4.13) gives $g(A)g(\emptyset) = g(A)$. Thus $g(A)$ should be 1, the identity for multiplication. Instead of g and h being numerically valued functions, they could be set valued functions and we could replace the summation in (4.11) with either a set union or a set intersection. (In terms of the previous notation, we would write A_i instead of $h(i)$.) Then $g(\emptyset)$ would be taken to be the identity for set union or set intersection respectively; that is, either $g(A) \cup g(\emptyset) = g(A)$ or $g(A) \cap g(\emptyset) = g(A)$. This leads to $g(\emptyset) = \emptyset$ and $g(\emptyset) = S$, respectively.

Let's recap where we were before our digression: We defined

$$f(A) \;=\; \left|\bigcup_{i\in A} S_i\right|$$

for $A \neq \emptyset$ and $f(\emptyset) = |S|$. Now, N_r is simply the sum of $f(R)$ over all subsets R of $\underline{m}$ of size r; i.e.,

$$N_r \;=\; \sum_{\substack{R\subseteq \underline{m}\\ |R|=r}} f(R).$$

We can rewrite (4.3) in the form

$$\sum_{R\subseteq \underline{m}} (-1)^{|R|} f(R).$$

In words, we can describe $f(\{i_1,\ldots,i_r\})$ as the number of things that satisfy conditions $i_1,\ldots,i_r$ and possibly others. In a similar manner, we could define a function $e(\{i_1,\ldots,i_r\})$ to the number of things that satisfy condition $i_1,\ldots,i_r$ and none of the other conditions in the collection $1,\ldots,m$. In these terms, (4.3) is a formula for $e(\emptyset)$. Also, by the definitions of e and f, we have $f(R) = \sum_{Q\supseteq R} e(Q)$.

We state without proof a generalization of the Principle of Inclusion and Exclusion.

Theorem 4.4 *Let P be the partially ordered set of subsets of $\underline{k}$. For any two functions e and f with domain P*

$$f(x) \;=\; \sum_{y\supseteq x} e(y) \quad \textit{for all } x\in P \tag{4.14}$$

if and only if

$$e(x) \;=\; \sum_{y\supseteq x} (-1)^{|y|-|x|} f(y) \quad \textit{for all } x\in P. \tag{4.15}$$

This result can be extended to any finite partially partially ordered set P if $(-1)^{|y|-|x|}$ is replaced by a function $\mu(x,y)$, called the Möbius function of the partially ordered set P. We will not explore this.

Exercises

4.1.12. This exercise extends the Principle of Inclusion and Exclusion (4.3). Let E_k be the number of elements of S that lie in exactly k of the sets $S_1,\ S_2,\ \ldots,\ S_m$. Prove that

$$E_k \;=\; \sum_{i=0}^{m-k} (-1)^i \binom{k+i}{i} N_{k+i}. \tag{4.16}$$

4.1.13. The purpose of this exercise is to prove Bonferroni's inequalities.

(a) Prove the inequalities are equivalent to the statement that sums

$$c_t(X) \;=\; \binom{|X|}{0} - \binom{|X|}{1} + \binom{|X|}{2} - \cdots \pm \binom{|X|}{t}$$

alternate in sign until eventually becoming 0 for $t \geq |X|$.

(b) Prove $c_t(X) \;=\; (-1)^t \binom{|X|-1}{t}$ and so complete the proof.

4.1.14. Find a formula that bears the same relation to Bonferroni's inequalities that (4.4) bears to (4.3); i.e., find inequalities for approximations to $|S_1\cup\cdots\cup S_m|$ rather than for approximations to E.

4.1.15. Consider the following algorithm due to H. Wilf.

Initialize: Let the N_i be defined as in (4.3).
Loop: Execute the following code.

```
For j = 0, 1, ..., m − 1
    For i = m − 1, m − 2, ..., j
        N_i = N_i − N_{i+1}
    End for
End for
```

The loop on i means that i starts at $m-1$ and decreases to j.

(a) By carrying out the algorithm for $m = 2$ and $m = 3$, prove that N_i is replaced by E_i, where E_i is given by Exercise 4.1.12 for these values of m.

(b) We can rephrase the algorithm in a set theoretic form. Replace N_r by N_r^*, the multiset which contains each $s \in S$ as many times as there are $1 \le i_1 < i_2 < \ldots < i_r \le m$ such that

$$s \in S_{i_1} \cap \ldots \cap S_{i_r}. \tag{4.17}$$

By the definition of N_r, it follows immediately that $N_r = |N_r^*|$. Similarly, replace E_k by E_k^*, the set of those elements of S that belong to exactly k of the sets $S_1, \ldots, S_m$. Replace N_i by N_i^* in the algorithm and interpret $N_i^* - N_{i+1}^*$ to be the multiset that contains $s \in S$ as many times as it appears in N_i^* minus the number of times it appears in N_{i+1}^*. We claim that the algorithm now stops with N_i^* replaced by E_i^*. Prove this for $m = 2$ and $m = 3$.

*(c) Using induction on t, prove the set theoretic form of the algorithm by proving that after t iterations of the loop on j; i.e., $j = 0, \ldots, t-1$ the following is true. If $s \in S$ appears in exactly p of the sets $S_1, \ldots, S_m$, then it appears in N_r^* with multiplicity

$$\mu(p,r,t) \;=\; \begin{cases} \binom{p-t}{r-t}, & \text{if } t \le p; \\ 1, & \text{if } t > p \text{ and } r = p; \\ 0, & \text{if } t > p \text{ and } r \ne p. \end{cases} \tag{4.18}$$

Also prove that no $s \in S$ ever appears more times in an N_{r+1}^* than it does in an N_r^* when we are calculating $N_r^* - N_{r+1}^*$.

(d) Prove that the validity of the set theoretic form of the algorithm implies the validity of the numerical form of the algorithm.
Hint. Use the last sentence in (c).

4.1.16. Let $D_n(k)$ be the number of permutations of $\underline{n}$ that have exactly k fixed points. Thus $D_n(0) = D_n$, the number of derangements of $\underline{n}$.

(a) Use Exercise 4.1.12 to obtain a formula for $D_n(k)$.

(b) Give a simple, direct combinatorial proof that $D_n(k) = \binom{n}{k} D_{n-k}$.

(c) Using algebra and (4.10), prove the answers in (a) and (b) are equal.

4.1.17. Let $A = \{a_1, \ldots, a_m\}$ be a set of m integers, all greater than 1. Let $d(n,k,A)$ be the number of integers in $\underline{n}$ that are divisible by exactly k of the integers in A.

(a) Assuming that the elements of A are distinct primes all dividing n, obtain a formula for $d(n,k,A)$ by using Exercise 4.1.12. Specialize this formula to obtain a formula for the Euler phi function $\varphi(n)$ discussed in Exercise 4.1.5.

(b) Relax the constraints in (a) by replacing the assumption that the elements in A are primes by the assumption that no two elements in A have a common factor.

(c) Relax the constraints in (a) further by not requiring that the elements of A divide n.

(d) Can you relax the constraints in (a) still further by making no assumptions about A and n except that A consists of m integers greater than 1?

4.1.18. Explain why the real numbers with $x \rho y$ meaning "x is less than y" is not a poset.

4.1.19. Prove that the following are posets.

(a) The real numbers with $x\rho y$ meaning "x is less than or equal to y."

(b) The real numbers with $x\rho y$ meaning "x is greater than or equal to y."

(c) The subsets of a set with $x\rho y$ meaning $x \subseteq y$.

(d) The positive integers with $x\rho y$ meaning y/x is an integer.

4.1.20. Prove that if (S, ρ) is a poset then so is (S, τ) where $x\tau y$ if and only if $y\rho x$.

4.1.21. Let S be the set of all partitions of $\underline{n}$. If $x, y \in S$, write $x\rho y$ if and only if every block of y is a union of one or more blocks of x. For example, $\{1,2\},\{3\},\{4\}\rho\{\{1,2,4\},\{3\}\}$. Prove that this is a poset.

4.1.22. Suppose that (R, ρ) and (T, τ) are posets. Prove that $(R \times T, \pi)$ is a poset if $(r, s)\pi(r', s')$ means that both $r\rho r'$ and $t\tau t'$ are true.

4.1.23. We will deduce the result in Exercise 4.1.12 as a consequence of the partially ordered set extension of the Principle of Inclusion and Exclusion.

(a) We look at subsets y of $\{1, 2, \ldots, m\}$. Let $e(y)$ be the number of elements in S that belong to every S_i for which $i \in y$ and to none of the S_j for which $j \notin y$. Prove that E_k is the sum of $e(y)$ over all y of size k.

(b) Prove that if $f(x)$ is defined by (4.14), then

$$f(x) \;=\; \Bigl|\bigcap_{i\in x} S_i\Bigr|.$$

(c) Conclude that (4.15) implies (4.16).

4.2 Listing Structures with Symmetries

By using decision trees, introduced in Chapter 3, we can produce our list C of canonical representatives. There are many ways to go about it. We'll illustrate this by some examples. Many of the examples are based on the Ferris wheel problem of Example 1.12 (p. 13): How many distinct six long circular sequences of ones and twos are there?

Example 4.7 A straightforward method One approach to the Ferris wheel problem is to simply generate all sequences and reject those that are equivalent to an earlier one in the lex order. For example, we would reject both 121121 and 211211 because they are equivalent to 112112, which occurs earlier in lex order.

We can reduce the size of the decision tree by being careful; e.g., the sequence that starts 1211 . . . can never be lexically least because we could shift it two positions to get 11 . . . 12.

Even with these ideas, the decision tree is rather large. Hence, we'll shorten the problem we've been considering to sequences of length 4. The decision tree is shown in Figure 4.2. It is simply the tree for generating all functions from $\underline{4}$ to $\underline{2}$ with those functions which have a (lexically) smaller circular shift removed. How did we do the removal? When we decided to begin with 2, there was no possibility of ever choosing a 1—a circular shift would begin with 1 and so be smaller. Also, if any 2's are present, we can never end with a 1 because a circular shift that moved it to the front would produce a smaller sequence. This rule was applied to determine the possible decisions at 112, 121 and 122. This explains everything that's missing from the full tree for $\underline{2}^{\underline{4}}$.

This approach can get rather unwieldy when doing larger problems by hand. Try using it for the six long Ferris wheel. □

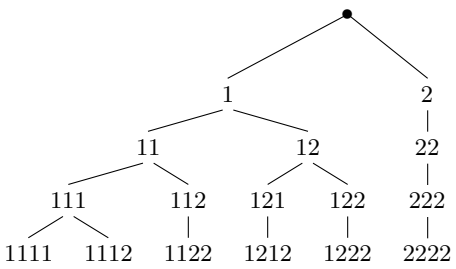


Figure 4.2 A Ferris wheel decision tree.

Example 4.8 Another problem Four identical spheres are glued together so that three of them lie at the vertices of an equilateral triangle and the fourth lies at the center. That is, the centers of the spheres lie in a plane and three of the centers are at the corners of an equilateral triangle while the fourth is in the center. Thus, the sphere arrangement remains unchanged in appearance if it is flipped over about any of three axes or if it is rotated 120 degrees about an axis that passes through the center of the center sphere and is perpendicular to the plane of the centers. Draw yourself a picture to illustrate this—it is very useful to get into the habit of drawing pictures to help visualize problems like this.

We have four tiny identical red balls and four tiny identical green balls. The balls are to be placed in the spheres so that each sphere contains exactly two balls. How many arrangements are possible?

The calculation can be done with the help of a decision tree. The first decision could be the number of red balls to be placed in the center sphere. If no red balls are placed in the center sphere, then two green balls must be placed there and two in the outer spheres. Those two in the outer spheres can either be placed in the same sphere or in different spheres. Proceeding in this sort of way, we can construct a decision tree. You should do this, verifying that exactly six distinct arrangements are possible. □

Example 4.9 A subtler method Another approach to the Ferris wheel problem is to take into account some of the effects of the symmetry when designing the decision tree. Let's look at our six long Ferris wheel.

The basic idea is to look at properties that depend only on the circular sequence rather than on how we have chosen to write it as a list. Unlike the simpler approach of listing things in lex order, there are a variety of choices for constructing the decision tree. As a result, different people may construct different decision trees. Constructing a good tree may take a fair bit of thought. Is it worth the effort? Yes, because a good decision tree may be considerably smaller than one obtained by a more simplistic approach.

Before reading further in this example, construct a simple lex order decision tree like the one in Figure 4.2, but for the six long Ferris wheel.

Since the number of ones in a sequence remains the same, we can partition the problem according to the number of ones that appear in the 6 long sequence. Thus our list of possible first decisions could be

more 1's than 2's, three of each, more 2's than 1's.

We can save ourselves some work right away by noting that all the sequences that arise from the third choice can be obtained from those of the first choice by replacing 1's with 2's and 2's with 1's.

What should our next decisions be? We'll do something different. Define a function s on sequences with $s(x)$ equal to the *minimal* amount the sequence x must be circularly shifted to obtain x again. (This is called the "period" of the circular sequence.) Thus $s(111111) = 1$, $s(121121) = 3$ and $s(122221) = 6$. Note that if x and y are equivalent, then $s(x) = s(y)$. You should convince yourself

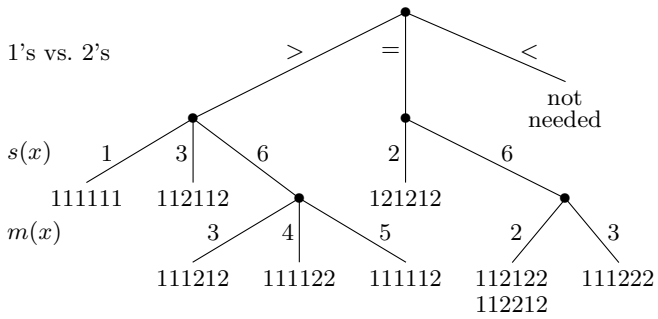


Figure 4.3 Another Ferris wheel decision tree.

that x consists of $6/s(x)$ copies of the first $s(x)$ elements of x. As a result, the only possible values of s are 1, 2, 3 and 6, and the ratio of 1's to 2's in x is the same as in the first $s(x)$ entries.

In this paragraph, we consider the case in which the number of 1's and 2's in x are equal. By the above, $s(x)$ must be even. If $s(x) = 2$, we need three repeats of a two long pattern that contains a 1 and a 2. We can take x to be either 121212 or 212121. We'll adopt our usual convention of using the lexically least equivalent sequence for the canonical sequence, so $x = 121212$. If $s(x) = 6$, the situation is more complex and so another decision will be used to break this case down further. Again, many choices are possible. We'll use $m(x)$, the length of the longest string of consecutive ones in the sequence x. Remember to read the list circularly, so $m(121211) = 3$. Put this sequence at the start of the list. A little thought should convince you that $m(x) = 1$ implies $x = 121212$, which does not have $s(x) = 6$. Since there are three 1's and three 2's, $m(x)$ must be 2 or 3. If $m(x) = 3$, we have $x = 111222$. If $m(x) = 2$, we have $x = 112??2$. The questionable entries must be a one and a two. Either order works giving us the two sequences 112122 and 112212.

A similar analysis to that in the previous paragraph can be used for the case in which there are more 1's than 2's. We leave it to you to carry out the analysis for this case.

The decision tree we have developed is shown in Figure 4.3. Compare its size with that of the simple lex order tree you were asked to construct. Construct another decision tree using a different sequence of decisions than we did in this example. Your goal should be to come up with something different from Figure 4.3 that is about the same size as it is. □

It is helpful to understand better the difference between the two methods we've used for the Ferris wheel problem. Our first method is a straightforward pruning of the decision tree for listing all functions in lex order. When several functions correspond to the same structure, we retain only the lexicographically least one of them. The simplicity of the method makes it fairly easy to program. Unfortunately it can lead to a rather large tree, often containing many decisions that lead to no canonical solutions. Thus, although it is straightforward, using it for hand calculation may lead to errors because of the amount of work involved.

Our second method requires some ingenuity. The basic idea is to select some feature of the problem that lets us break it into smaller problems of the same basic type but with additional conditions. Let's look at what we did. First we divided the problem into three parts depending on the number of ones versus the number of twos. Each part was a problem of the same type; e.g., how many different arrangements are there *where each arrangement has more ones than twos.* Isn't this what we did in the first method when we made a decision like "the first element of the sequence is 1?" No! This is not a problem of the same type because the condition is not invariant under rotation of the sequence. On the other hand, the condition that there be more ones than twos is invariant under rotation.

In our second method, we next chose another property that is invariant under rotation of the sequence: how much we had to rotate the sequence before it looked the same. Next we looked at the

longest consecutive string of ones, with the sequence read circularly so that the first entry follows the last. Again, this is invariant under rotation. Sometimes we did not need to go that far because it was easy to see the solutions; e.g., after $s(x) = 1$, it was clear that 111111 was the only solution. On the other hand, after the decision sequence $=, 6, 2$ in Figure 4.3, it was still not obvious what the answer was. At this time we decided it was easier to shift back to our first method rather than find another property that was invariant under rotation. We did this on scratch paper and simply wrote the result as two solutions in the figure.

We might call the second method the *symmetry invariant* method. Why is symmetry invariance better than the first method? When done cleverly, it leads to smaller decision trees and hence less chance for computational errors. On the other hand, you may make mistakes because it is less mechanical or you may make poor selections for the decision criteria. If you are applying symmetry invariance, how do you decide what properties to select as decision criteria? Also, how do you decide when to switch back to the first method? There are no rules for this. Experience is the best guide.

Example 4.10 Listing necklaces We'll work another example using symmetry invariance. How many ways can the corners of a regular hexagon be labeled using the labels B, R and W, standing for the colors blue, red and white? Note that an unlabeled hexagon can be rotated 60° and/or flipped over and still look the same. You could imagine this as a hexagon made from wire with a round bead to be placed at each corner. We impose a condition on the finished hexagon of beads:

Adjacent beads must be different colors.

Our first decision will be the number of colors that actually appear. If only two are used, there are only three solutions: BRBRBR, BWBWBW and RWRWRW since adjacent colors must be different. (We've used the same sort of notation we used for the Ferris wheel.) If three colors are used, we decide how many of each actually appear. The possibilities are 2,2,2 and all six permutations of 1,2,3. To do the latter, we need only consider the case of 1 blue, 2 red and 3 white and then permute the colors in our solutions in all six possible ways. (To count, we simply multiply this case by 6.)

Let's do the 1 blue, 2 red and 3 white case by the first method. A canonical sequence must start with B and be followed somehow by 2 R's and 3 W's so that adjacent letters are different. Call the sequence associated with the hexagon $\mathrm{B}x_2x_3x_4x_5x_6$. There are no solutions with $x_2 = \mathrm{R}$ or with $x_6 = \mathrm{R}$ because we are not allowed to have two W's adjacent. Thus $x_2 = x_6 = \mathrm{W}$. We easily obtain the single solution BWRWRW. Remember that this gives six solutions through permutation of colors.

The case in which each color is used twice remains to be done. We make a decision based on whether the two B's are opposite each other or not on the hexagon. We use the first method now. The case of opposite B's leads to the sequence $\mathrm{B}x_2x_3\mathrm{B}x_5x_6$ and the other case leads to $\mathrm{B}y_2\mathrm{B}y_4y_5y_6$. In the first case, x_2 and x_3 are different. This leads to two lexically least sequences: BRWBRW and BRWBWR. (The sequence BWRBWR is just BRWBRW flipped over.) In the second case, choosing y_2 determines the remaining y's. The two results are BRBWRW and BWBRWR.

Adding up our results, there are $3 + 6 \times 1 + (2 + 2) = 13$ solutions. □

Exercises

4.2.1. Redo Example 4.10 using only the first (mechanical) method.

4.2.2. How many ways can the eight corners of a regular octagon be labeled using the labels B and W. Note that an unlabeled octagon can be rotated 45° and/or flipped over and still look the same.

4.2.3. Let $F(r)$ the number of ways to place beads at the vertices of a square when we are given r different types of round beads. Let $f(r)$ be the same number except that at least one bead of each of the r types must be used. Rotations and reflections of the square are allowed as with the hexagon in Example 4.10.

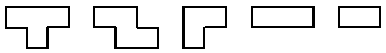
(a) Prove that $F(r) = \binom{r}{1}f(1) + \binom{r}{2}f(2) + \binom{r}{3}f(3) + \binom{r}{4}f(4)$

(b) By evaluating $f(r)$ for $r \leq 4$, obtain an explicit formula for $F(r)$.

4.2.4. State and prove a generalization of the formula in the previous exercise that expresses $F(r)$ in terms of the function f. Possible generalizations are to the hexagon and the n-gon. Can you find a generalization that has little or no connection with symmetries?

4.2.5. We want to list the coverings of a 4×4 board by 8 dominoes, where solutions that differ only by a rotation and/or reflection of the board are considered to be the same. For example, Figure 3.15 shows 11 ways to cover a 3×4 board. With rotation and/or reflection, only 5 are distinct. The lex order minimal descriptions of the distinct ones are *hhhhhh*, *hhhvvh*, *hhvhvh*, *hhvvv* and *hvvvvh*. List the lexically least coverings of the 4×4 board; i.e., our standard choices for canonical representatives.

4.2.6. Draw a decision tree for covering the 4×4 board using one each of the shapes shown below. Two boards are equivalent if one can be transformed to the other by rotations and/or reflections.
Hint. First place the "T" shaped piece.



4.2.7. This problem is concerned with listing colorings of the faces of a cube. Unless you are very good at visualizing in three dimensions, we recommend that you have a cube available to manipulate. (Even a sugar cube could be used.) Also, when listing solutions, you may find it convenient to represent the cube in the plane by "unfolding" it as shown and writing the colors in the squares. The line of four faces can be thought of as the four sides and the other two faces can be thought as the top and bottom.

(a) List and count the ways to color the faces using at most the 2 colors black and white.

(b) List and count the ways to color the faces using at most the two colors black and white, with the added condition that we do not distinguish between a cube and its color negative (interchanging black and white).

(c) List and count the ways to color the faces using at most the two colors black and white, with the added condition we cannot distinguish between a cube and its mirror image.

(d) List and count the ways to color the faces using all of the colors black, red and white; i.e., every color must appear on each cube.

(e) Count the ways to color the faces using the colors black, red and white. On any given cube, all colors need not appear.

(f) Find a formula $F(r)$ for the number of ways to color the faces of a cube using r colors so that whenever two faces are opposite one another they are colored differently.
Hint. See Exercise 4.2.4.

4.2.8. We say that f is a "Boolean" function if $f\colon \{0,1\}^n \to \{0,1\}$.

(a) Prove that a Boolean function with $n = 2$ can be thought of as placing zeroes and ones at the corners of a 1×1 square with lower left corner at the origin. Give a similar interpretation for $n = 3$ using a cube.

(b) We want to count the number of "different" Boolean functions with $n = 2$ and $n = 3$. Two functions will be considered equivalent if one can be obtained from the other by permuting the arguments and/or complementation. We can describe this precisely in an algebraic fashion by saying that $f, g\colon \{0,1\}^n \to \{0,1\}$ are equivalent if and only if there is a permutation σ of $\underline{n}$ and $c, d_1, \ldots, d_n \in \{0,1\}$ such that

$$f(x_1, \ldots, x_n) \;=\; c \oplus g(x_{\sigma(1)} \oplus d_1, \ldots x_{\sigma(n)} \oplus d_n),$$

where $u \oplus v$ is $u + v$ unless $u = v = 1$ in which case $u \oplus v = 0$. ("Exclusive or" and "mod 2 sum" are other names for $u \oplus v$.) For $n = 2$, there are four different Boolean functions:

$$\big(f(0,0), f(0,1), f(1,0), f(1,1)\big) \;=\; \begin{cases} (0,0,1,1), & (0,1,0,1), \\ (0,1,1,1), & (1,1,1,,1). \end{cases}$$

Interpret the equivalence of Boolean functions in terms of symmetries involving the square and cube when $n = 2, 3$.

(c) List the different Boolean functions when $n = 3$.

4.3 Counting Structures with Symmetries

We've been using "equivalent" rather loosely without saying what it means. Since ambiguous terms provide an easy way to make errors, we should define it.

Definition 4.2 Equivalence *An* **equivalence relation** *on a set S is a partition of S. We say that $s, t \in S$ are* **equivalent** *if and only if they belong to the same block of the partition. If the symbol $\sim$ denotes the equivalence relation, then we write $s \sim t$ to indicate that s and t are equivalent. An* **equivalence class** *is a subset of S consisting of all objects in the set that are equivalent to some object; i.e., an equivalence class is a block of the partition.*

Returning to our circular sequence problem, what do the equivalence classes look like? First, 111111 is in a class by itself because all rotations give us the same sequence again. Likewise, 222222 is in a class by itself. The sequences {121212, 212121} is a third equivalence class. The sequences 112112 and 122122 are in different equivalence classes, each of which contains 3 sequences. So far, we have 5 equivalence classes containing a total of 10 sequences. What about the remaining $2^6 - 10 = 54$ sequences? It turns out that they fall into 9 equivalence classes of 6 sequences each. Thus there are $5 + 9 = 14$ equivalence classes; that is, the answer to our circular sequence problem is 14.

This method is awkward for larger problems. You might try to do 12 long circular sequences of ones, twos and threes, where the answer is 44,368. Burnside's Lemma allows us to do such problems more easily. In order to state and prove it we need some observations about the symmetries.

In our problem the symmetries are rotations of the Ferris wheel through 0°, 60°, 120°, 180°, 240° and 300°. These correspond to reading a sequence circularly starting with the first, second, ... and sixth positions, respectively. Let S be the set of all six long sequences of zeroes and ones. The six symmetries correspond to six permutations of S by means of the circular reading. For example, if g_i is the permutation that starts reading in position $i + 1$, then $g_1(111122) = 111221$ and $g_3(111122) = 122111$. Note that $g_0(x) = x$ for all x. Alternatively, we can think of g_i as shifting the sequence "circularly" to the left by i positions. The set $G = \{g_0, \ldots, g_5\}$ has some important properties, namely

(G-1) There is an $e \in G$ such that $e(x) = x$ for all $x \in S$.

(G-2) If $f \in G$, then the inverse of f exists and $f^{-1} \in G$.

(G-3) If $f, g \in G$, then the composition fg is in G.

The function e is called the "identity" and e is reserved for its name. You should be able to verify that $g_i^{-1} = g_{5-i}$ and $g_i g_j = g_k$, where $k = i + j$ if this is less than 6 and $k = i + j - 6$ otherwise. Any set of permutations with properties (G-1), (G-2) and (G-3) is called a *permutation group*. Group theory is an important subject that is part of the branch of mathematics called algebra. We barely touch on it here.

Symmetries always lead to permutation groups. Why? First, recall that a symmetry of some thing is a rearrangement that leaves the thing looking the same (in our case, the thing is the empty Ferris wheel). Taking the inverse corresponds to reversing the motion of the symmetry, so it again leaves the thing looking the same. Taking a product corresponds to one symmetry followed by another and so leaves the thing looking the same.

What is the connection between the equivalence classes and the permutation group for the sequences? It is simple: Two sequences $x, y \in S$ are equivalent if and only if $y = g(x)$ for some $g \in G$. In general, a group G of permutations on a set S defines an equivalence relation in this way. That requires a bit of proof, which we give at the end of the section. We can now state Burnside's Lemma, but we defer its proof until the end of the section. In this theorem the expression $\sum_{g \in G} N(g)$ appears. For those unfamiliar with such notation, it means that we must add up the values of $N(g)$ for all $g \in G$. The order in which we add them does not matter since order is irrelevant in addition.

Theorem 4.5 Burnside's Lemma *Let S be a set with a permutation group G. The number of equivalence classes that G defines on S is*

$$\frac{1}{|G|} \sum_{g \in G} N(g),$$

where $N(g)$ is the number of $x \in S$ such that $g(x) = x$.

Example 4.11 The Ferris wheel generalized We'll redo the Ferris wheel problem and generalize it.

Burnside's Lemma tells us that the answer to the Ferris wheel problem is

$$\tfrac{1}{6}\big(N(g_0) + N(g_1) + N(g_2) + N(g_3) + N(g_4) + N(g_5)\big).$$

Let's compute the terms in the sum. $N(g_0) = 2^6$ since $g_0 = e$, the identity, and there are two choices for each of the six positions in the sequence. What is $N(g_1)$? If $x = x_1x_2x_3x_4x_5x_6$, then $g_1(x) = x_2x_3x_4x_5x_6x_1$. Since we want $g_1(x) = x$, we need $x_1 = x_2$, $x_2 = x_3$, ... and $x_6 = x_1$. In other words, all the x_i's are equal. Thus $N(g_1) = 2$. Since $g_2(x) = x_3x_4x_5x_6x_1x_2$, we find that $x_1 = x_3 = x_5$ and $x_2 = x_4 = x_6$. Thus $N(g_2) = 2^2 = 4$. You should be able to prove that $N(g_3) = 8$, $N(g_4) = 4$ and $N(g_5) = 2$. Thus the number of equivalence classes is $\frac{1}{6}(64 + 2 + 4 + 8 + 4 + 2) = 14$.

Now suppose that instead of placing just ones and twos in circular sequences, we have k symbols to choose from. Our work in the previous paragraph makes it easy for us to write down a formula. Note that for $g_2(x) = x$ we found that $x_1 = x_3 = x_5$ and $x_2 = x_4 = x_6$. Thus we can choose one symbol as the value for $x_1 = x_3 = x_5$ and another (possibly the same) symbol as the value for $x_2 = x_4 = x_6$. Thus $N(g_2) = k^2$. The other $N(g_i)$ values can be determined in a similar manner giving us

$$\tfrac{1}{6}(k^6 + k + k^2 + k^3 + k^2 + k) \;=\; \frac{k(k^5 + k^2 + 2k + 2)}{6}$$

different arrangements. When $k = 2$, we obtain the result from the previous paragraph.

Now let's modify what we just did by adding the requirement that adjacent symbols must be different. In this case, $N(g_1) = 0$ because $g_1(x) = x$ requires that $x_1 = x_2$, which is forbidden. For

$g_3(x) = x$, we have $x_1 = x_4$, $x_2 = x_5$ and $x_3 = x_6$. Since the symbols assigned to $x_1 = x_4$, x_2 and x_3 must all be different, $N(g_3) = k(k-1)(k-2)$. With a bit more work, we find that there are

$$\tfrac{1}{6}\big(N(g_0) + 0 + k(k-1) + k(k-1)(k-2) + k(k-1) + 0\big)$$

equivalence classes. The determination of $N(g_0)$ is a bit more difficult. We'll discuss this type of problem in Section 6.2. The answer is $N(g_0) = (k-1)^6 + k - 1$. Thus the number of equivalence classes is

$$\tfrac{1}{6}(k-1)\big((k-1)^5 + k^2 + 1\big).$$

We can check these calculations a bit by noting that when $k = 1$ there should be no solutions and when $k = 2$ there should be 1. Substitution into our formula does indeed give 0 and 1. □

We can shorten some of the work in the previous example by thinking of permutations a bit differently. Instead of looking at permutations of 6 long sequences, we can look at the way the permutation rearranges the positions of the sequence. For example, $g_2(x_1x_2x_3x_4x_5x_6) = x_3x_4x_5x_6x_1x_2$ can be interpreted as saying position 1 is replaced by position 3, position 2 is replaced by position 4, and so forth. This is a new permutation: Instead of permuting the set of 6 long sequences it permutes the set $\underline{6}$. To emphasize both the difference and the relationship, we use γ, the Greek letter corresponding to g. In cycle form, $\gamma_2 = (1,3,5)(2,4,6)$. Thinking of a sequence as a function $f\colon \underline{6} \to \underline{k}$, the function will be counted by $N(\gamma_2) = N(g_2)$ if and only if it is constant on the cycles of γ_2. This is a general result:

> **Principle** *In the set of allowed functions, $N(\gamma)$ counts precisely those allowed functions which are constant on the cycles of γ; i.e., those functions f such that $f(x) = f(y)$ whenever x and y are in the same cycle of γ.*

For example, if all functions $f\colon A \to B$ are allowed, $N(\gamma)$ is simply $|B|^c$ where c is the number of cycles of γ. In the concluding problem of the last example, not all functions were allowed because the adjacency constraint requires that $f(i) \neq f(i+1)$ for all i. In that case, computing $N(\gamma)$ is a bit harder, but the principle can still be used.

Example 4.12 Counting necklaces How many ways can 4 identical round green beads and 4 identical round red beads be arranged to form a necklace of 8 beads? Due to the nature of the beads, two necklaces will be the same if one can be obtained from the other by rotation or flipping over. The symmetries are like those in Example 4.10 (p. 109) except that we now have 8 positions instead of 6.

We can imagine a necklace as an 8 long sequence and use the idea we just discussed to describe the permutations. Obviously all that matters for counting purposes is the size of the cycles—not what they contain. Altogether there are 16 permutations of $\underline{8}$, which are shown here in 5 classes, where P_k is the set of permutations having k cycles. We've omitted commas separating entries in the cycles. We leave it to you to check that the list is correct and complete.

P_8	(1)(2)(3)(4)(5)(6)(7)(8)		
P_5	(1)(2,8)(3,7)(4,6)(5)	(1,3)(2)(4,8)(5,7)(6)	(1,5)(2,4)(3)(6,8)(7)
	(1,7)(2,6)(3,5)(4)(8)		
P_4	(1,5)(2,6)(3,7)(4,8)	(1,2)(3,8)(4,7)(5,6)	(1,4)(2,3)(5,8)(6,7)
	(1,6)(2,5)(3,4)(7,8)	(1,8)(2,7)(3,6)(4,5)	
P_2	(1,3,5,7)(2,4,6,8)	(1,7,5,3)(2,8,6,4)	
P_1	(1,2,3,4,5,6,7,8)	(1,4,7,2,5,8,3,6)	(1,8,7,6,5,4,3,2)
	(1,6,3,8,5,2,7,4)		

Suppose that $\gamma \in P_8$; i.e., $\gamma = (1)\cdots(8)$. Since each cycle has length 1, we can simply choose 4 cycles to be the green beads. This can be done in $\binom{8}{4} = 70$ ways. Thus $N(\gamma) = 70$. Suppose $\gamma \in P_5$. We can either choose 2 of the 2 cycles to be green beads OR choose both 1 cycles and one of the 2 cycles. Thus $N(\gamma) = \binom{3}{2} + \binom{3}{1} = 6$. Similarly, for P_4, P_2 and P_1 we have the values $\binom{4}{2} = 6$, 2 and 0, respectively, for $N(\gamma)$. Thus the number of necklaces is

$$\tfrac{1}{16}(70 + 6 \times 4 + 6 \times 5 + 2 \times 2) \;=\; 8.$$

Since there were only a few solutions, it probably would have been easier to count them by first listing them. Do it. Unfortunately, it is usually not easy to tell in advance that there will be so few solutions that listing them is easier than using Burnside's Lemma. One approach is to start listing them. If there seem to be too many, your time will probably not have been wasted because you will have gotten a better feel for the problem. □

*Proofs

We'll conclude this section with the two proofs that we put off:

1. A permutation group G on a set S gives an equivalence relation on S.
2. Burnside's Lemma is true.

Our proofs will be fairly heavy in notation and manipulation. If this causes difficulties for you, it may help if you consider what is happening in a simple case. For example, you might look the permutations associated with the Ferris wheel problem. You may also need to reread the proofs.

Proof: (Permutation groups give equivalence relations.) To prove this, we must prove that defining $x, y \in S$ to be equivalent if and only if $y = g(x)$ for some $g \in G$ does indeed give an equivalence relation. In other words, we must prove that there is a partition of S such that x and y are in the same block of S if and only if $y = g(x)$ for some $g \in G$.

Let

$$B_x \;=\; \{y \in S | y = g(x) \text{ for some } g \in G\}. \tag{4.19}$$

We need to know that the set of B_x's form a partition of S.

(a) We have $x \in B_x$ because $x = e(x)$. Thus every x in S is in at least one block.

(b) We must prove that the blocks are disjoint; that is, if $B_x \cap B_y \neq \emptyset$, then $B_x = B_y$. Suppose that $z \in B_x \cap B_y$ and $w \in B_x$, then, by (4.19), there are permutations f, g and h such that $z = f(x)$, $z = g(y)$ and $w = h(x)$. Thus

$$w \;=\; h(x) \;=\; h(f^{-1}(z)) \;=\; h(f^{-1}(g(y))) \;=\; (hf^{-1}g)(y).$$

By (G-2) and (G-3), $hf^{-1}g \in G$. Thus $w \in B_y$. We proved that $B_x \subseteq B_y$. Similarly, $B_y \subseteq B_x$ and so $B_x = B_y$. □

Proof: (Burnside's Lemma) Before proving Burnside's Lemma, we'll prove something that will be needed later in the proof. Let x be some element of S and let I_x be the set of all $g \in G$ such that $g(x) = x$. We will prove that

$$|I_x| \cdot |B_x| \;=\; |G|, \qquad 4.20 \quad \text{where } B_x \text{ is defined by (4.19).}$$

To illustrate this, consider our Ferris wheel problem with $x = 121212$, we have $B_x = \{121212, 212121\}$ and $I_x = \{g_0, g_2, g_4\}$ and $|G| = 6$. You should look at some other examples to convince yourself that (4.20) is true in general.

How can we prove (4.20)? We use a trick. Let $F\colon G \to S$ be defined by $F(g) = g(x)$. Be careful: x is *fixed* and g is the variable. Note that $\text{Image}(F) = B_x$ and $F^{-1}(x) = I_x$. We claim that $|F^{-1}(y)| = |F^{-1}(x)|$ for all $y \in B_x$. The validity of this claim is enough to prove (4.20) because the claim proves that the coimage of F, which is a partition of G, consists of $|B_x|$ blocks each of size $|F^{-1}(x)|$.

We now prove the claim. Now both x and y are fixed. Since $y \in B_x$, there is some $h \in G$ such that $y = h(x)$. Then

$$\begin{aligned}
F^{-1}(y) &= \{g \in G \mid g(x) = y\} && \text{by the definition of } F^{-1};\\
&= \{g \in G \mid g(x) = h(x)\} && \text{since } y = h(x);\\
&= \{g \in G \mid (h^{-1}g)(x) = x\} && \text{by (G-2);}\\
&= \{hk \mid k \in G \text{ and } k(x) = x\} && \text{by setting } k = h^{-1}g;\\
&= \{hk \mid k \in F^{-1}(x)\} && \text{by the definition of } F;\\
&= hF^{-1}(x).
\end{aligned}$$

This gives us a bijection between $F^{-1}(y)$ and $F^{-1}(x)$. Thus $|F^{-1}(y)| \;=\; |F^{-1}(x)|$.

We now prove Burnside's Lemma. The number of equivalence classes is simply the number of distinct B_x's. Unfortunately we can't easily get our hands on an entire equivalence class or a canonical representative. The following observation will let us look at all the elements in each equivalence class; i.e., all the elements in S.

$$\text{For any set } T, \quad 1 \;=\; \sum_{t \in T} \frac{1}{|T|}.$$

You should be able to prove this easily.

Let $\mathcal{E}$ be the set of equivalence classes of S. Then

$$|\mathcal{E}| \;=\; \sum_{B \in \mathcal{E}} 1 \;=\; \sum_{B \in \mathcal{E}} \sum_{y \in B} \frac{1}{|B|} \;=\; \sum_{B \in \mathcal{E}} \sum_{y \in B} \frac{1}{|B_y|},$$

since $B_y = B$. The last double sum is just $\sum_{y \in S} 1/|B_y|$ because each $y \in S$ belongs to exactly one equivalence class. Let $\chi(P)$ be 1 if the statement P is true and 0 if it is false. This is called a *characteristic function*. Using the above and (4.20),

$$\begin{aligned}
|\mathcal{E}| &= \sum_{y \in S} \frac{1}{|B_y|} \;=\; \sum_{y \in S} \frac{|I_y|}{|G|} \;=\; \frac{1}{|G|} \sum_{y \in S} |I_y|\\
&= \frac{1}{|G|} \sum_{y \in S} \Big(\sum_{g \in G} \chi(g(y) = y) \Big) && \text{by the definition of } I_y;\\
&= \frac{1}{|G|} \sum_{g \in G} \Big(\sum_{y \in S} \chi(g(y) = y) \Big) && \text{by interchanging summation;}\\
&= \frac{1}{|G|} \sum_{g \in G} N(g) && \text{by the definition of } N(g).
\end{aligned}$$

This completes the proof of Burnside's Lemma. □

Exercises

4.3.1. Suppose that you can count only ordered lists and you would like a formula for $C(n,k)$, the number of k element subsets of $\underline{n}$. Let A be the set of all k-lists without repeated elements that can be formed from $\underline{n}$. Let B be all subsets of $\underline{n}$. We define $F\colon A \to B$ as follows. For $a \in A$, let $F(a)$ be the set whose elements are the items in the list a. By studying the image of F and $|F^{-1}(x)|$ for $x \in \text{Image}(F)$, obtain a formula for $C(n,k)$.

4.3.2. How many 8-long circular sequences can be made using the ten digits $0, 1, 2, \ldots, 9$ if no digit can appear more than once? Can you generalize your answer to n-long circular sequences when k things are available instead of just ten?

4.3.3. Redo Example 4.12 with the numbers of beads changed from 4 and 4 to 3 and 5. Use Burnside's Lemma.

4.3.4. Redo Example 4.12 where there are k colors of beads and there are no constraints on how often a color may be used.

4.3.5. Label the vertices of a regular n-gon clockwise using the numbers 1 to n in order. We can describe a symmetry of the n-gon by a permutation of $\underline{n}$. The set of n symmetries of the n-gon that involve just rotating it in the plane about its center is called the *cyclic group* on $\underline{n}$. The set that also allows flipping the n-gon over is called the *dihedral group* on $\underline{n}$. It contains $2n$ symmetries, including the original n from the cyclic group.

(a) Describe the elements of the cyclic group as permutations in two line form. (There is a very simple description of the second line. You should be able to find it by drawing a picture and rotating it.)

(b) Describe the elements of the dihedral group as permutations in two line form. (There is a simple description of the second line of the additional permutations not in the cyclic group.)

(c) Describe the cycles of the elements of the dihedral group that are not in the cyclic group.

4.3.6. How many ways can 8 squares be colored green on a 4×4 board of 16 squares?

(a) Assume that the only symmetries that are allowed are rotations of the board.

(b) Assume that the board can be rotated and flipped over.

4.3.7. Starting with the observation

$$\sum_{y\in S}\Big(\sum_{g\in G}\chi(g(y)=y)\Big) \;=\; \sum_{g\in G}\Big(\sum_{y\in S}\chi(g(y)=y)\Big),$$

use (4.20) to prove Burnside's Lemma. (This is just a rearrangement of the proof in the text.)

4.3.8. Let $D(n)$ be the number of ways to arrange n dominoes to cover a $2 \times n$ board with no symmetries allowed. Let $d(n)$ be the number of ways to arrange them when rotations and reflections are allowed.

(a) List the coverings that give $D(5) = 8$ and $D(6) = 13$. Describe the coverings in general.

(b) Prove that $D(n)$ is the number of compositions of n where the only allowed parts are 1 and 2.

(c) Prove that $d(n)$ is the number of equivalence classes of compositions of n into ones and twos, where two compositions are equivalent if they are the same or if reading one from left to right is the same as the other read from right to left.

(d) Prove that

$$d(n) \;=\; \begin{cases} \frac{1}{2}\Big(D(n)+D(k)\Big) & \text{if } n = 2k+1; \\ \frac{1}{2}\Big(D(n)+D(k)+D(k-1)\Big) & \text{if } n = 2k. \end{cases}$$

Notes and References

Alternative discussions of the Principle of Inclusion and Exclusion can be found in the texts by Bogart [3; Ch.3], Stanley [6; Ch.2] and Tucker [7; Ch.8]. Stanley [6; Sec.2.6] uses the "Involution Principle" to give a "bijective" proof of the Principle of Inclusion and Exclusion. A bijective proof of a formula first interprets both sides of a formula as counting something in a simple manner (in particular, no minus signs are normally present). The proof consists of a bijection between the two sets of objects being counted. The Involution Principle is a fairly new technique for proving bijections that was introduced by Garsia and Milne [4]. Exercise 4.1.15 was adapted from [8].

Gian-Carlo Rota [5] introduced Möbius inversion to combinatorialists. A less advanced discussion of Möbius inversion and its applications has been given by Bender and Goldman [1]. Möbius inversion is only one of the many aspects of partially ordered sets which have become important in combinatorial theory. See Stanley [6; Ch.3] for an introduction. In turn, partially ordered sets are only one of the tools of modern algebraic mathematics that are important in combinatorics. This explosive growth of algebraic methods in combinatorics began in the late 1960's.

We'll return to the study of objects with symmetries in Section 11.3, where we connect a special case of Burnside's Lemma with generating functions. Among the texts that discuss enumeration with symmetries are those by Biggs [2; Chs.13,14] and Tucker [7; Ch.9]. Williamson [9; Ch.4] goes deeper into some aspects related to computer applications. The study of objects with symmetries is inevitably tied to the theory of permutation groups, which we have attempted to minimize. See Biggs [2] for more background on group theory.

1. Edward A. Bender and Jay R. Goldman, On the applications of Möbius inversion in combinatorial analysis, *American Math. Monthly* **82** (1975), 789–803.
2. Norman L. Biggs, *Discrete Mathematics*, 2nd ed., Oxford Univ. Press (2003).
3. Kenneth P. Bogart, *Introductory Combinatorics*, 3rd ed., Brooks/Cole (2000).
4. Adriano M. Garsia and Stephen C. Milne, A Rogers-Ramanujan bijection, *J. Combinatorial Theory, Series A* **31** (1981), 289–339.
5. Gian-Carlo Rota, On the foundations of combinatorial theory I. Theory of Möbius functions, *Zeitschrift für Wahrscheinlichkeitstheorie* **2** (1964), 340–368.
6. Richard P. Stanley, *Enumerative Combinatorics*, vols. 1 and 2, Cambridge Univ. Press (1999, 2001).
7. Alan C. Tucker, *Applied Combinatorics*, 4th ed., John Wiley (2001).
8. Herbert S. Wilf, Two algorithms for the sieve method, *J. of Algorithms* **12** (1991), 179–182.
9. S. Gill Williamson, *Combinatorics for Computer Science*, Dover (2002).

PART II
Graphs

Graph theory is one of the most widely applicable areas of mathematics. Its concepts and terminology are used in many areas to help formulate and clarify ideas. Graph theory theorems find application in a wide range of fields, particularly the newer scientific disciplines.

The notion of a "graph" is deceptively simple: It is a collection of points (called "vertices") that are joined by lines (called "edges"). Often all that matters about the edges is which two vertices they join, not their length or how they curve. This concept is deceptive because it seems unlikely that such a simple, general notion could have an interesting theory or be of any use. Simplicity is important. Scientists often try to find the simplest workable solution both in formulating theories and designing experiments. Mathematicians try to find the simplest concepts that usefully encompass what they are studying. A computer scientist, being part scientist and part mathematician, also looks for simplicity. Why is there this push for simplicity?

1. Many scientists believe that the underlying laws of the universe should exhibit elegance and simplicity.
2. The more complicated a construction is, the more likely it is to malfunction. Some familiar examples of this are experimental apparatus, algorithms and computer programs.
3. A simple concept is usually more flexible than a complex one and so can be applied in more new situations. Common examples in which simplicity is a virtue are definitions, scientific theories and data structures.

"Simple" should not be confused with "unsophisticated." Special relativity, complex variables and context–free grammars are all simple; but none of them are unsophisticated.

We'll introduce some of the basic concepts in graph theory in Chapter 5 and then discuss some theory and applications in Chapter 6. To thoroughly discuss applications of graphs in computer science would require a very large book. Another would be needed to discuss the purely mathematical aspects of graph theory. We've picked a variety of important topics from different areas of graph theory and computer science. The topics we've chosen reflect our perceptions of what you should learn and also our own interests. These topics are

- Spanning trees: an important tool in combinatorial algorithms;
- Graph coloring: a subject with pretty results and a relatively long history;
- Planarity: a deep subject with connections to graph coloring;
- Flows in networks: an important application of graphs;
- Random graphs: the properties of typical graphs;
- Finite state machines: an important concept for formal languages and compiler design.

CHAPTER 5

Basic Concepts in Graph Theory

Introduction

The concepts in this chapter are essential for understanding later discussions involving graphs, so be sure that you understand them. It is not necessary to memorize all the concepts since you can refer back to them if necessary; however, make sure that you understand them when you study them now so that referring back to them will simply be a memory refresher, not a new learning experience.

Since the basic concepts in Section 2.1 (p. 41) are used in this chapter, you may wish to review them before continuing.

5.1 What is a Graph?

There are various types of graphs, each with its own definition. Unfortunately, some people apply the term "graph" rather loosely, so you can't be sure what type of graph they're talking about unless you ask them. After you have finished this chapter, we expect you to use the terminology carefully, not loosely. To motivate the various definitions, we'll begin with some examples.

Example 5.1 A computer network Computers are often linked with one another so that they can interchange information. Given a collection of computers, we would like to describe this linkage in fairly clean terms so that we can answer questions such as "How can we send a message from computer A to computer B using the fewest possible intermediate computers?"

We could do this by making a list that consists of pairs of computers that are connected. Note that these pairs are unordered since, if computer C can communicate with computer D, then the reverse is also true. (There are sometimes exceptions to this, but they are rare and we will assume that our collection of computers does not have such an exception.) Also, note that we have implicitly assumed that the computers are distinguished from each other: It is insufficient to say that "A PC is connected to a Mac." We must specify which PC and which Mac. Thus, each computer has a unique identifying label of some sort.

For people who like pictures rather than lists, we can put dots on a piece of paper, one for each computer. We label each dot with a computer's identifying label and draw a curve connecting two

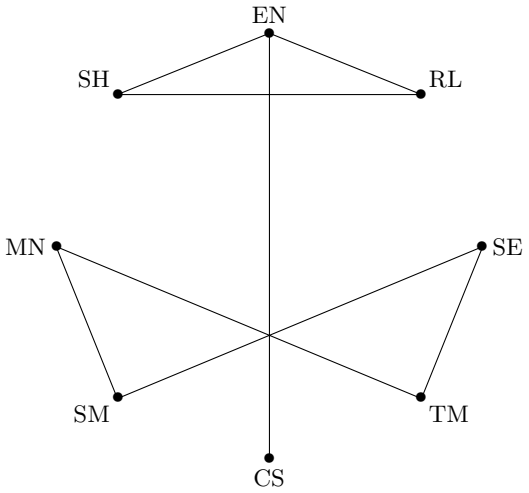


Figure 5.1 Computers connected by networks. Computers (vertices) are indicated by dots (•) with labels. The connections (edges) are indicated by lines. When lines cross, they should be thought of as cables that lie on top of each other—not as cables that are joined.

dots if and only if the corresponding computers are connected. Note that the shape of the curve does not matter (it could be a straight line or something more complicated) because we are only interested in whether two computers are connected or not. Figure 5.1 shows such a picture. Each computer has been labeled by the initials of its owner.

Recall that $\mathcal{P}_2(V)$ stands for the set of all two element subsets of the set V. Based on our computer example we have

Definition 5.1 Simple graph *A* **simple graph** *G is a set V, called the* **vertices** *of G, and a subset E of $\mathcal{P}_2(V)$ (i.e., a set E of 2 element subsets of V), called the* **edges** *of G. We can represent this by writing $G = (V, E)$.*

In our case, the vertices are the computers and a pair of computers is in E if and only if they are connected. □

Example 5.2 Routes between cities Imagine four cities named, with characteristic mathematical charm, A, B, C and D. Between these cities there are various routes of travel, denoted by a, b, c, d, e, f and g. A picture of this situation is shown in Figure 5.2. Looking at it, we see that there are three routes between cities B and C. These routes are named d, e and f. Figure 5.2 is intended to give us a picture of only the interconnections between cities. It leaves out many aspects of the situation that might be of interest to a traveler. For example, the nature of these routes (rough road, freeway, rail, etc.) is not portrayed. Furthermore, unlike a typical map, no claim is made that the picture represents in any way the distances between the cities or their geographical placement relative to each other. The object shown in Figure 5.2 is called a *graph*. Edges a and b are called *parallel*, as are edges d, e and f.

Following our previous example, one is tempted to list the pairs of cities that are connected; in other words, to extract a simple graph from the information. Unfortunately, this does not describe the problem adequately because there can be more than one route connecting a pair of cities; e.g., d, e and f connecting cities B and C in the figure. How can we deal with this? Definition 5.2 is a precise definition of a graph of the type required to handle this type of problem.

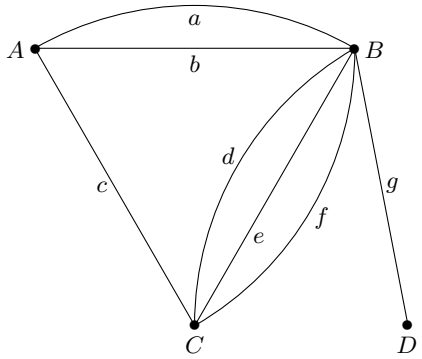


Figure 5.2 Capital letters indicate cities and lower case indicate routes.

Definition 5.2 Graph *A* **graph** *is a triple $G = (V, E, \varphi)$ where V and E are finite sets and φ is a function with range $\mathcal{P}_2(V)$ and with domain E. We call E the set of* **edges** *of the graph G. The set V is called the set of* **vertices** *of G.*

In Figure 5.2, $G = (V, E, \varphi)$ where

$$V = \{A, B, C, D\}, \quad E = \{a, b, c, d, e, f, g\}$$

and

$$\varphi = \begin{pmatrix} a & b & c & d & e & f & g \\ \{A,B\} & \{A,B\} & \{A,C\} & \{B,C\} & \{B,C\} & \{B,C\} & \{B,D\} \end{pmatrix}.$$

Definition 5.2. tells us that to specify a graph G it is necessary to specify the sets V and E and the function φ. We have just specified V and φ in *set theoretic* terms. Figure 5.2 specifies the same V and φ in pictorial terms. The set V is represented clearly in Figure 5.2 by dots ($\bullet$), each of which has a city name adajcent to it. Similarly, the set E is also represented clearly. The function φ is determined from Figure 5.2 by comparing the name attached to a route with the two cities connected by that route. Thus, the route name d is attached to the route with endpoints B and C. This means that $\varphi(d) = \{B, C\}$.

Note that, since part of the definition of a function includes its range and domain, φ determines $\mathcal{P}_2(V)$ and E. Also, V can be determined from $\mathcal{P}_2(V)$. Consequently, we could have said that a graph is a function φ whose domain is a finite set and whose range is $\mathcal{P}_2(V)$ for some finite set V. Instead, we choose to specify V and E explicitly because the vertices and edges play a fundamental role in thinking about a graph G. □

The function φ is sometimes called the *incidence function* of the graph. The two elements of $\varphi(x) = \{u, v\}$, for any $x \in E$, are called the vertices of the edge x, and we say u and v are *joined* by x. We also say that u and v are *adjacent vertices* and that u is *adjacent to* v or , equivalently, v is adjacent to u. For any $v \in V$, if v is a vertex of an edge x then we say x is *incident* on v. Likewise, we say v is a member of x, v is on x, or v is in x. Of course, v is a member of x actually means v is a member of $\varphi(x)$.

Figure 5.3 shows two other pictorial ways of specifying the same graph as in Figure 5.2. The drawings look very different but exactly the same set V and function φ are specified in each case. It is *very important* that you understand exactly what information is needed to completely specify the graph. When thinking in terms of cities and routes between them, you naturally want the pictorial representation of the cities to represent their geographical positioning also. If the pictorial representation does this, that's fine, but it is not a part of the information required to define a graph. Geographical location is extra information. The geometrical positioning of the vertices A, B, C and D is very different in Figures 5.2 and 5.3(a). However, in each of these cases, the vertices on a given

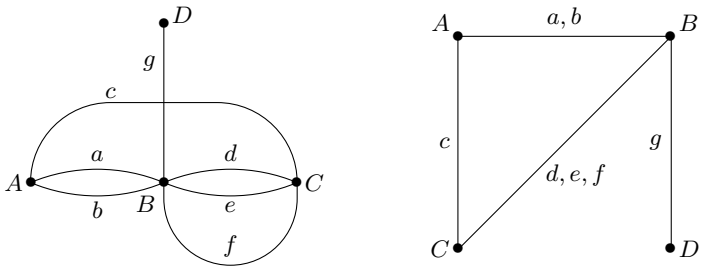


Figure 5.3 Two alternate pictorial specifications of Figure 5.2.

edge are the same and hence the graphs specified are the same. In Figure 5.3(b) a different method of specifying the graph is given. There, φ^{-1}, the inverse of φ, is given. For example, $\varphi^{-1}(\{C, B\})$ is shown to be $\{d, e, f\}$. Knowing φ^{-1} determines φ and hence determines G since the vertices A, B, C and D are also specified in Figure 5.3(b).

Warning *Some people call our "simple graph" a "graph" and some people call our "graph" a "multigraph." Still other people mean something somewhat different by the term multigraph!*

Example 5.3 Simple graphs are graphs We can easily reconcile our two definitions by realizing that a simple graph is a special case of a graph. Let $G = (V, E)$ be a simple graph. Define $\varphi: E \to E$ to be the identity map; i.e., $\varphi(e) = e$ for all $e \in E$. The graph $G' = (V, E, \varphi)$ is essentially the same as G. There is one subtle difference in the pictures: The edges of G are unlabeled but each edge of G' is labeled by a set consisting of the two vertices at its ends. □

There are still more concepts that can be called graphs. People may not be interested in which road is which in Figure 5.2, so the labels on the edges are not needed. On the other hand, some people might need more information, such as how long it takes to travel on each of the routes in Figure 5.2. There may be other situations, too; for example, some of the routes may be one way. We will meet some of these concepts later.

Exercises

5.1.1. Let (V, E, φ) be a graph and $v \in V$ a vertex. Define the *degree* of v, $d(v)$ to be the number of $e \in E$ such that $v \in \varphi(e)$; i.e., e is incident on v. Prove that $\sum_{v \in V} d(v) = 2|E|$, an even number. Conclude that the number of vertices v for which $d(v)$ is odd is even.

5.1.2. We are interested in the number of simple graphs with $V = \underline{n}$.

(a) Prove that there are $2^{\binom{n}{2}}$ such simple graphs. (That's 2 to the power $\binom{n}{2}$, not $2\binom{n}{2}$.)

(b) How many of them have exactly q edges?

(c) If we choose a simple n-vertex graph uniformly at random, what is the probability that it has exactly q edges?

5.1.3. Sometimes it is useful to allow an edge to have both its ends on the same vertex. Let $Q = (V, E, \varphi)$ be a graph where

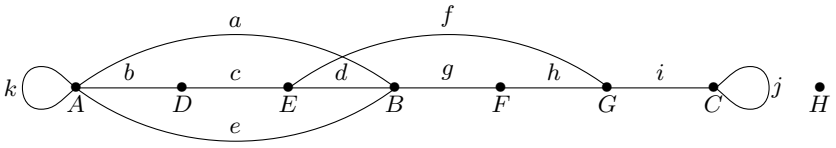
$$V = \{A, B, C, D, E, F, G, C, H\}, \quad E = \{a, b, c, d, e, f, g, h, i, j, d\}$$

and

$$\varphi = \begin{pmatrix} a & b & c & d & e & f & g & h & i & j & k \\ A & A & D & E & A & E & B & F & G & C & A \\ B & D & E & B & B & G & F & G & C & C & A \end{pmatrix}.$$

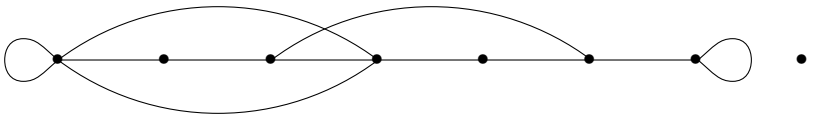
In this representation of φ, the first row specifies the edges and the the two vertices below each edge specify the vertices incident on that edge. Here is a pictorial representation $P(Q)$ of this graph.

$P'(Q)$:



Note that $\varphi(k) = \{A, A\} = \{A\}$. Such an edge is called a *loop*. Adding a loop to a vertex increases its degree by two. The vertex H, which does not belong to $\varphi(x)$ for any edge x (i.e., has no edge incident upon it), is called an *isolated* vertex. The degree of an isolated vertex is zero. Edges, such as a and e of Q, with the property that $\varphi(a) = \varphi(e)$ are called *parallel* edges. If all edge and vertex labels are removed from $P(Q)$ then we get the following picture $P'(Q)$:

$P'(Q)$:



The picture $P'(Q)$ represents the "form" of the graph just described and is sometimes referred to as a pictorial representation of the "unlabeled" graph associated with Q. For each of the following graphs R, where $R = (V, E, \varphi)$, $V = \{A, B, C, D, E, F, G, C, H\}$, draw a pictorial representation of R by starting with $P'(Q)$ removing and/or adding as few edges as possible and then labeling the resulting picture with the edges and vertices of R. A graph R which require no additions or removals of edges is said to be "of the same form as" or "isomorphic to" the graph Q.

(a) Let $E = \{a, b, c, d, e, f, g, h, i, j, k\}$ be the set of edges of R and

$$\varphi = \begin{pmatrix} a & b & c & d & e & f & g & h & i & j & k \\ C & C & F & A & H & E & E & A & D & A & A \\ C & G & G & H & H & H & F & H & G & D & F \end{pmatrix}.$$

(b) Let $E = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$ be the set of edges of R and

$$\varphi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ A & E & E & E & F & G & H & B & C & D & E \\ G & H & E & F & G & H & B & C & D & D & H \end{pmatrix}.$$

5.1.4. Let $Q = (V, E, \varphi)$ be a graph with $|V| = n$. Let $d_1, d_2, \ldots, d_n$, where $d_1 \leq d_2 \leq \cdots \leq d_n$ be the sequence of degrees of the vertices of Q, sorted by size. We refer to this sequence as the *degree sequence* of the graph Q. For example, if $Q = (V, E, \varphi)$ is the graph where

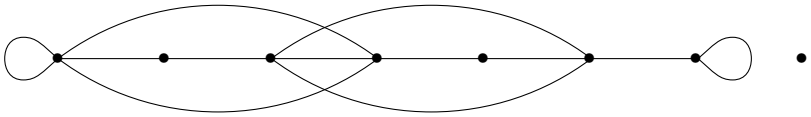
$$V = \{A, B, C, D, E, F, G, H\}, \quad E = \{a, b, c, d, e, f, g, h, i, j, k, l\}$$

and

$$\varphi = \begin{pmatrix} a & b & c & d & e & f & g & h & i & j & k & l \\ A & A & D & E & A & E & B & F & G & C & A & E \\ B & D & E & B & B & G & F & G & C & C & A & G \end{pmatrix}.$$

then $(0, 2, 2, 3, 4, 4, 4, 5)$ is the degree sequence of Q. Consider the the following unlabeled pictorial representation of Q

$P'(Q)$:



(a) Create a pictorial representation of Q by labeling $P'(Q)$ with the edges and vertices of Q.

(b) A necessary condition that a pictorial representation of a graph R can be created by labeling $P'(Q)$ with the vertices and edges of R is that the degree sequence of R be $(0, 2, 2, 3, 4, 4, 4, 5)$. True of false? Explain.

(c) A sufficient condition that a pictorial representation of a graph R can be created by labeling $P'(Q)$ with the vertices and edges of R is that the degree sequence of R be $(0, 2, 2, 3, 4, 4, 4, 5)$. True or false? Explain.

5.1.5. In each of the following problems information about the degree sequence of a graph is given. In each case, decide if a graph, *without loops*, satisfying the specified conditions exists or not. Give reasons in each case.

(a) A graph Q with degree sequence $(1, 1, 2, 3, 3, 5)$?

(b) A graph Q with degree sequence $(1, 2, 2, 3, 3, 5)$, multiple (i.e. parallel) edges allowed?

(c) A simple graph Q with degree sequence $(1, 2, 2, 3, 3, 5)$?

(d) A simple graph Q with degree sequence $(3, 3, 3, 3)$?

(e) A graph Q with degree sequence $(3, 3, 3, 3)$, no loops or parallel edges allowed?

(f) A graph Q with degree sequence $(3, 3, 3, 5)$, no loops or parallel edges allowed?

(g) A graph Q with degree sequence $(4, 4, 4, 4, 4)$, no loops or parallel edges allowed?

(h) A graph Q with degree sequence $(4, 4, 4, 4, 6)$, no loops or parallel edges allowed?

5.2 Equivalence Relations and Unlabeled Graphs

Sometimes we are interested only in the "structure" of a graph and not in the names (labels) of the vertices and edges. In this case we are interested in what is called an unlabeled graph. A picture of an unlabeled graph can be obtained from a picture of a graph by erasing all of the names on the vertices and edges. This concept is simple enough, but is difficult to use mathematically because the idea of a picture is not very precise.

The concept of an *equivalence relation* on a set is an important concept in mathematics and computer science. We used the idea in Section 4.3, but did not discuss it much there. We'll explore it more fully here and will use it to rigorously define unlabeled graphs. Later we will use it to define connected components and biconnected components. We recall the definition given in Section 4.3:

Definition 5.3 Equivalence relation *An* **equivalence relation** *on a set S is a partition of S. We say that $s, t \in S$ are* **equivalent** *if and only if they belong to the same block. If the symbol $\sim$ denotes the equivalence relation, then we write $s \sim t$ to indicate that s and t are equivalent.*

Example 5.4 To refresh your memory, we'll look at some simple equivalence relations.

Let S be any set and let all the blocks of the partition have one element. Two elements of S are equivalent if and only if they are the same. This rather trivial equivalence relation is, of course, denoted by "=".

Now let the set be the integers $\mathbb{Z}$. Let's try to define an equivalence relation by saying that n and k are equivalent if and only if they differ by a multiple of 24. Is this an equivalence relation? If it is we should be able to find the blocks of the partition. There are 24 of them, which we could number $0, \ldots, 23$. Block j consists of all integers which equal j plus a multiple of 24; that is, they have a remainder of j when divided by 24. Since two numbers belong to the same block if and only if they both have the same remainder when divided by 24, it follows that they belong to the same block if and only if their difference gives a remainder of 0 when divided by 24, which is the same as saying their difference is a multiple of 24. Thus this partition does indeed give the desired equivalence relation.

Now let the set be $\mathbb{Z} \times \mathbb{Z}^*$, where $\mathbb{Z}^*$ is the set of all integers except 0. Write $(a, b) \sim (c, d)$ if and only if $ad = bc$. With a moment's reflection, you should see that this is a way to check if the two fractions a/b and c/d are equal. We can label each equivalence class with the fraction a/b that it represents. In an axiomatic development of the rationals from the integers, one defines a rational number to be just such an equivalence class and proves that it is possible to add, subtract, multiply and divide equivalence classes. We won't pursue this. □

In the next theorem we provide necessary and sufficient conditions for an equivalence relation. Verifying the conditions is a useful way to prove that some particular situation is an equivalence relation. Recall that a *binary relation* on a set S is a subset R of $S \times S$. Given a binary relation R, we will write $s \sim t$ if and only if $(s, t) \in R$. Thus "$\sim$" is another way to represent the binary relation.

Theorem 5.1 Equivalence relations *Let S be a set and suppose that we have a binary relation $\sim$ on S. This is an equivalence relation if and only if the following three conditions hold.*

(*i*) *(Reflexive) For all $s \in S$ we have $s \sim s$.*

(*ii*) *(Symmetric) For all $s, t \in S$ such that $s \sim t$ we have $t \sim s$.*

(*iii*) *(Transitive) For all $r, s, t \in S$ such that $r \sim s$ and $s \sim t$ we have $r \sim t$.*

Proof: We first prove that an equivalence relation satisfies (i)–(iii). Suppose that $\sim$ is an equivalence relation. Since s belongs to whatever block it is in, we have $s \sim s$. Since $s \sim t$ means that s and t belong to the same block, we have $s \sim t$ if and only if we have $t \sim s$. Now suppose that $r \sim s \sim t$. Then r and s are in the same block and s and t are in the same block. Thus r and t are in the same block and so $r \sim t$.

We now suppose that (i)–(iii) hold and prove that we have an equivalence relation. What would the blocks of the partition be? Everything equivalent to a given element should be in the same block. Thus, for each $s \in S$ let $B(s)$ be the set of all $t \in S$ such that $s \sim t$. We must show that the set of these sets form a partition of S.

In order to have a partition of S, we must have

(a) every $t \in S$ is in some $B(s)$ and

(b) for every $p, q \in S$, $B(p)$ and $B(q)$ are either equal or disjoint.

Since $\sim$ is reflexive, $s \in B(s)$, proving (a). Suppose $x \in B(p) \cap B(q)$ and $y \in B(p)$. We have, $p \sim x$, $q \sim x$ and $p \sim y$. Thus $q \sim x \sim p \sim y$ and so $y \in B(q)$, proving that $B(p) \subseteq B(q)$. Similarly $B(q) \subseteq B(p)$ and so $B(p) = B(q)$. This proves (b). □

Suppose we have a picture of a graph $G = (V, E, \varphi)$, with the elements of V and E written next to the appropriate vertices and edges in the picture. Suppose that we have another graph $G' = (V', E', \varphi')$. We may be able to erase the elements of V and E and replace them with elements of V' and E', respectively, so that we obtain a picture of the graph G'. If this is possible, we will say that G and G' are isomorphic graphs. One can show that this relation (G is isomorphic to G') satisfies the conditions of Theorem 5.1 and so is an equivalence relation. All graphs which are isomorphic to a given graph correspond to the same unlabeled graph. The following definition and example formulate these ideas more precisely.

Definition 5.4 Graph isomorphism *Let $G = (V, E, \varphi)$ and $G' = (V', E', \varphi')$ be graphs. We say G and G' are* **isomorphic**, *written $G \sim G'$ if there are bijections*

$$\nu: V \to V' \quad \text{and} \quad \varepsilon: E \to E'$$

such that $\varphi'(\varepsilon(e)) = \nu(\varphi(e))$ for all $e \in E$, where $\nu(\{x, y\})$ is defined to be $\{\nu(x), \nu(y)\}$.

Let $G = (V, E)$ and $G' = (V', E')$ be simple graphs. They are isomorphic if there is a bijection $\nu: V \to V'$ such that

$$\{u, v\} \in E \quad \textit{if and only if} \quad \{\nu(u), \nu(v)\} \in E'.$$

Let's see what our definition means intuitively. Suppose we have a picture of a graph $G = (V, E, \varphi)$, with the elements of V and E written next to the appropriate vertices and edges in the picture. We can replace the vertex set by a new set V', writing the elements of V' where the elements of V were. The same thing can be done with the edges and a new set E'. This defines two functions $\nu: V \to V'$ and $\varepsilon: E \to E'$. The condition $\varphi'(\varepsilon(e)) = \nu(\varphi(e))$ says that we get φ' by simply looking at the picture to see what the ends of an edge are.

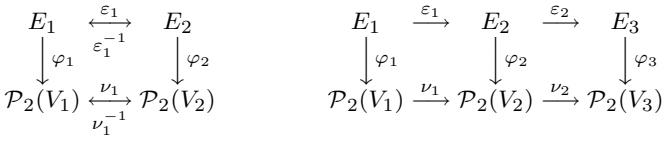


Figure 5.4 Functions involved in proving the equivalence of graphs. The left diagram is used for the reflexive law. The right diagram is used for the transitive law.

Example 5.5 Unlabeled graphs Let S be the set of all graphs and let $G = (V, E, \varphi)$ and $G' = (V', E', \varphi')$ be in S. We will prove that graph isomorphism is an equivalence relation on S.

Before we give our proof, let's look at how an equivalence class can be interpreted. Draw a picture of G and then erase the names of the vertices and edges. We could call what is left the "structure" of G or, as is more commonly done, an *unlabeled graph*. The use of an equivalence relation to define an unlabeled graph may seem a bit round-about: Why not just be satisfied with the picture? Thinking about the picture is fine for thinking about unlabeled graphs and for the proofs we'll write; however, there are reasons for presenting this definition:

- If we wanted to use the picture approach in a more formal way, we'd have to say precisely what a picture of a graph was and when two pictures represented the same graph. We faced this problem a bit in the previous section.
- Equivalence relations arise from isomorphisms in many areas of mathematics and there is often no picture that one can use to describe the equivalence relation. Thus, seeing the formalism for graphs is good preparation for seeing it in other courses.
- If we wanted a theorem-proving computer program to work with unlabeled graphs, we'd need to give it a formal definition. (It would be much simpler to work with labeled graphs.) This is an example of how an intuitively simple concept—unlabeled graphs in this case—can be very difficult to express in terms that computer software can deal with.

We now give a formal proof that we have an equivalence relation.

First: $G \sim G$ because we can take ν and ε to be the identity functions; i.e., $\nu(x) = x$ for all $x \in V$.

Second: Given that $G_1 \sim G_2$, we must prove $G_2 \sim G_1$, the reflexive law. Let ν_1 and ε_1 be the bijections guaranteed by the definition of $G_1 \sim G_2$. Then $\nu_1^{-1}\colon V_2 \to V_1$ and $\varepsilon_1^{-1}\colon E_2 \to E_1$ are bijections. Let $e_2 \in E_2$ and $e_1 = \varepsilon_1^{-1}(e_2)$. Then, by $\varphi_2(\varepsilon_1(e_1)) = \nu_1(\varphi_1(e_1))$, we have $\varphi_1(e_1) = \nu_1^{-1}(\varphi_2(\varepsilon_1(e_1)))$ and so

$$\varphi_1(\varepsilon_1^{-1}(e_2)) \;=\; \varphi_1(e_1) \;=\; \nu_1^{-1}(\varphi_2(\varepsilon_1(e_1))) \;=\; \nu_1^{-1}(\varphi_2(e_2)).$$

Thus $G_2 \sim G_1$.

Figure 5.4 may help you follow what we've just done: The definition of $G_1 \sim G_2$ says that starting at $e \in E_1$, going to E_2 via ε_1 and then to $\mathcal{P}_2(V_2)$ via φ_2 ends up at the same pair of vertices as going from e to $\mathcal{P}_2(V_1)$ and then to $\mathcal{P}_2(V_2)$ via φ_1 and ν_1, respectively. In other words, the two routes from E_1 to $\mathcal{P}_2(V_2)$ end up in the same place. We proved that the two routes from E_2 to $\mathcal{P}_2(V_1)$ must then also end up in the same place.

Third: We must prove the transitive law. Suppose that $G_1 \sim G_2 \sim G_3$. We then have the bijections $\nu_i\colon V_i \to V_{i+1}$ and $\varepsilon_i\colon E_i \to E_{i+1}$ for $i = 1, 2$. Furthermore, $\varphi_{i+1}(\varepsilon_i(e_i)) = \nu_i(\varphi_i(e_i))$ for all $e_i \in E_i$. Let $\nu(v_1) = \nu_2(\nu_1(v_1))$ and $\varepsilon(e_1) = \varepsilon_2(\varepsilon_1(e_1))$. It is easily verified that ν and ε are bijections since ν_i and ε_i are. Finally, since $\varepsilon_1(e_1) \in E_2$,

$$\varphi_3(\varepsilon(e_1)) \;=\; \varphi_3(\varepsilon_2(\varepsilon_1(e_1))) \;=\; \nu_2(\varphi_2(\varepsilon_1(e_1))) \;=\; \nu_2(\nu_1(\varphi_1(e_1))) \;=\; \nu(\varphi_1(e_1)).$$

Thus $G_1 \sim G_3$. □

Exercises

5.2.1. Suppose that $G = (V, E, \varphi)$ and $G' = (V', E', \varphi')$ are equivalent; i.e., give the same unlabeled graph. Prove the following.

(a) $|V| = |V'|$ and $|E| = |E'|$.

(b) $d(k, G) = d(k, G')$ for all k, where $d(k, H)$ is the number of vertices of degree k in H.

5.2.2. From our discussion, it may seem to be an easy matter to decide if two graphs represent the same unlabeled graph. This is not true even for relatively small graphs. Divide the following graphs into equivalence classes and justify your answer; i.e., explain why you have the classes that you do. In all cases $V = \underline{4}$.

(a) $\varphi = \begin{pmatrix} a & b & c & d & e & f \\ \{1,2\} & \{1,2\} & \{2,3\} & \{3,4\} & \{1,4\} & \{2,4\} \end{pmatrix}$

(b) $\varphi = \begin{pmatrix} A & B & C & D & E & F \\ \{1,2\} & \{1,4\} & \{1,4\} & \{1,2\} & \{2,3\} & \{3,4\} \end{pmatrix}$

(c) $\varphi = \begin{pmatrix} u & v & w & x & y & z \\ \{2,3\} & \{1,3\} & \{3,4\} & \{1,4\} & \{1,2\} & \{1,2\} \end{pmatrix}$

(d) $\varphi = \begin{pmatrix} P & Q & R & S & T & U \\ \{3,4\} & \{2,4\} & \{1,3\} & \{3,4\} & \{1,2\} & \{1,2\} \end{pmatrix}$

5.2.3. Let $M(n, \mathbb{R})$ be the $n \times n$ matrices over the real numbers.

(a) For two matrices $A, B \in M(n, \mathbb{R})$, write $A \simeq B$ if and only if there is some nonsingular $P \in M(n, \mathbb{R})$ such that $B = PAP^{-1}$. Prove that this is an equivalence relation.

(b) For two matrices $A, B \in M(n, \mathbb{R})$, write $A \sim B$ if and only if there is some nonsingular $P \in M(n, \mathbb{R})$ such that $B = PAP^t$, where P^t is the transpose of P. Prove that this is an equivalence relation.

5.2.4. Which of the following define equivalence relations? If an equivalence relation is not defined, why not?

(a) For all s and t, $s \sim t$.

(b) Among the students at a university who have selected precisely one major, two are equivalent if and only if they have the same major.

(c) Among the students at a university, two students are equivalent if they have a class in common.

(d) For the real numbers, two numbers are equivalent if they differ by less that 0.001.

(e) For the real numbers, two numbers are equivalent if they agree in their decimal expansions through the third digit after the decimal place.

*5.2.5. Define the concept of a equivalence relation for simple graphs so that you can introduce the notion of an unlabeled simple graph. Prove that you have, indeed, defined an equivalence relation.
Hint. You need only introduce ν, not ε.

*5.2.6. We say that an ordinary function $f \in B^A$ has its domain A and its range B *labeled*. For convenience,we let $A = \{1, \ldots, a\}$ and $B = \{1, \ldots, b\}$.

(a) Suppose that $f, g \in B^A$. Write $f \sim g$ if there is a permutation π on A such that $f(x) = g(\pi(x))$ for all $x \in A$. Prove that this is an equivalence relation. We call the equivalence class a function with unlabeled domain. Prove that the set of nondecreasing functions from A to B is a system of representatives for these equivalence classes; that is, this set contains exactly one function from each equivalence class.

(b) Using the idea in (a), define the notion of a function with unlabeled range and prove that you have an equivalence relation. Call a function $f\colon A \to B$ a "restricted growth function" if $f(1) = 1$ and, for $a > j \geq 1$, $f(j+1)$ is at most one larger than the maximum of $f(1)$, $f(2)$, $\ldots$, $f(j)$. Prove that the restricted growth functions form a system of representatives for the equivalence classes you have defined.

(c) Using the previous ideas, define the notion of a function with unlabeled domain and range and prove that you have an equivalence relation. Call a function $f\colon A \to B$ a "partition function" if $f(i) \leq f(i+1)$ for $a > i \geq 1$ and $|f^{(-1)}(j)| \geq |f^{(-1)}(j+1)|$ for $b > j \geq 1$. Prove that the partition functions give one representative from each equivalence class

*5.2.7. This problem uses the ideas and notation from the previous problem. Construct a table with four rows marked with the four possibilities "A (un)labeled and B (un)labeled" and with the columns marked with "all," "injections" and "surjections." Each of the twelve positions is to be interpreted as the number of (equivalence classes of) functions in B^A satisfying the conditions. We use $|A| = a$ and $|B| = b$ and use U and L to indicate labeled and unlabeled. The start of a table is shown below. Verify these entries and complete the table. How can the number of (equivalence classes of) bijections be found from the table?

A	B	all	injections	surjections
L	L	?	$b(b-1)\cdots(b-a+1)$	?
L	U	$\sum_{k \leq b} S(a,k)$	?	?
U	L	?	?	$\binom{a-1}{a-b}$
U	U	?	1	?

5.3 Paths and Subgraphs

An important concept for describing the structure of a graph is the concept of a path.

Definition 5.5 Path, trail, walk and vertex sequence *Let $G = (V, E, \varphi)$ be a graph. Let $e_1, e_2, \ldots, e_{n-1}$ be a sequence of elements of E (edges of G) for which there is a sequence $a_1, a_2, \ldots, a_n$ of distinct elements of V (vertices of G) such that $\varphi(e_i) = \{a_i, a_{i+1}\}$ for $i = 1, 2, \ldots, n-1$. The sequence of edges $e_1, e_2, \ldots, e_{n-1}$ is called a* **path** *in G. The sequence of vertices $a_1, a_2, \ldots, a_n$ is called the* **vertex sequence** *of the path. (Note that since the vertices are distinct, so are the edges.) If we require that $e_1, \ldots, e_{n-1}$ be distinct, but not that $a_1, \ldots, a_n$ be distinct, the sequence of edges is called a* **trail**. *If we do not even require that the edges be distinct, it is called a* **walk**.

Note that the definition of a path requires that it not intersect itself (i.e., have repeated vertices), while a trail may intersect itself. Although a trail may intersect itself, it may not have repeated edges, but a walk may. If $P = (e_1, \ldots, e_{n-1})$ is a path in $G = (V, E, \varphi)$ with vertex sequence $a_1, \ldots, a_n$ then we say that P is a *path from* a_1 *to* a_n. Similarly for a trail or a walk.

In the graph of Figure 5.2 (p. 123), the sequence c, d, g is a path with vertex sequence A, C, B, D. If the graph is of the form $G = (V, E)$ with $E \subseteq \mathcal{P}_2(V)$, then the vertex sequence alone specifies the sequence of edges and hence the path. Thus, in Figure 5.1 (p. 122), the vertex sequence MN, SM, SE, TM specifies the path {MN, SM}, {SM, SE}, {SE, TM}.

Note that every path is a trail and every trail is a walk, but not conversely. However, we can show that, if there is a walk between two vertices, then there is a path. This rather obvious result can be useful in proving theorems, so we state it as a theorem.

Theorem 5.2 *Suppose $u \neq v$ are vertices in $G = (V, E, \varphi)$. The following are equivalent:*

(a) *There is a walk from u to v.*

(b) *There is a trail from u to v.*

(c) *There is a path from u to v.*

Furthermore, given a walk from u to v, there is a path from u to v all of whose edges are in the walk.

Proof: Since every path is a trail, (c) implies (b). Since every trail is a walk, (b) implies (a). Thus it suffices to prove that (a) implies (c). Let $e_1, e_2, \ldots, e_k$ be a walk from u to v. We use induction on n, the number of repeated vertices in a walk. If the walk has no repeated vertices, it is a path. This starts the induction at $n = 0$. Suppose $n > 0$. If u and or v is repeated, take the part of the walk that starts at the last occurrence of u and ends at the first occurrence of v, since this walk has less than n repeated vertices, there is a path. Let r be a repeated vertex different from u and v. Suppose it first appears in edge e_i and last appears in edge e_j. Then $e_1, \ldots, e_i, e_j, \ldots, e_k$ is a walk from u to v in which r is not a repeated vertex. Hence there are less than n repeated vertices in this walk from u to v and so there is a path by induction. Since we constructed the path by removing edges from the walk, the last statement in the theorem follows. □

Another basic notion is that of a subgraph of $G = (V, E, \varphi)$, which we will soon define. First we need some terminology about functions. By a *restriction* φ' of φ to $E' \subseteq E$, we mean the function φ' with domain E' and satisfying $\varphi'(x) = \varphi(x)$ for all $x \in E'$.

Definition 5.6 Subgraph *Let $G = (V, E, \varphi)$ be a graph. A graph $G' = (V', E', \varphi')$ is a* **subgraph** *of G if $V' \subseteq V$, $\varphi(e') \in \mathcal{P}_2(V')$ for all $e' \in E'$, and φ' is the restriction of φ to E' having range $\mathcal{P}_2(V')$.*

The fact that G' is itself a graph means that $\varphi(x) \in \mathcal{P}_2(V')$ for each $x \in E'$.

Example 5.6 For the graph $G = (V, E, \varphi)$ of Figure 5.2 (p. 123), Let $G' = (V', E', \varphi')$ be defined by $V' = \{A, B, C\}$, $E' = \{a, b, c, f\}$, and by φ' being the restriction of φ to E' with range $\mathcal{P}_2(V')$. Notice that φ' is determined completely from knowing V', E' and φ. Thus, to specify a subgraph G', the key information is V' and E'.

As another example from Figure 5.2, we let $V' = V$ and $E' = \{a, b, c, f\}$. In this case, the vertex D is not a member of any edge of the subgraph. Such a vertex is called an *isolated vertex* of G'. □

One way of specifying a subgraph is to give a set of edges $E' \subseteq E$ and take V' to be the set of all vertices on some edge of E'. In other words , V' is the union of the sets $\varphi(x)$ over all $x \in E'$. Such a subgraph is called the *subgraph induced by* E'. The first of Examples 5.6 is the subgraph induced by $E' = \{a, b, c, f\}$. Likewise, given a set $V' \subseteq V$, we can take E' to be the set of all edges $x \in E$ such that $\varphi(x) \subseteq V'$. The resulting subgraph is called the *subgraph induced by* V'. Referring to Figure 5.2 (p. 123), the edges of the subgraph induced by $V' = \{C, B\}$, are $E' = \{d, e, f\}$.

Look again at Figure 5.2. In particular, consider the path c, a with vertex sequence C, A, B. Notice that the edge d has $\varphi(d) = \{C, B\}$. The subgraph $G' = (V', E', \varphi')$, where $V' = \{C, A, B\}$ and $E' = \{c, a, d\}$ is called a *cycle* of G. In general, whenever there is a path in G, say $e_1, \ldots, e_{n-1}$ with vertex sequence $a_1, \ldots, a_n$, and an edge x with $f(x) = \{a_1, a_n\}$, then the subgraph induced by the edges $e_1, \ldots, e_{n-1}, x$ is called a *cycle* of G. The formal definition is:

Definition 5.7 Cycle *Let $G = (V, E, \varphi)$ be a graph and let $e_1, \ldots, e_{n-1}$ be a path with vertex sequence $a_1, \ldots, a_n$. If x is an edge of G such that $\varphi(x) = \{a_1, a_n\}$, then the subgraph G' of G induced by the set of edges $\{e_1, \ldots, e_{n-1}, x\}$ is called a* **cycle** *of G. The* **length** *of the cycle is n.*

In our definitions, a path is a *sequence* of edges but a cycle is a *subgraph* of G. In actual practice, we will not need to make such fine distinctions, so we may think of a cycle as a path, except that it starts and ends at the same vertex. Cycles are closely related to the existence of unique paths between vertices:

Theorem 5.3 *Two vertices $u \neq v$ are on a cycle of G if and only if there are two paths from u to v that have no vertices in common except the endpoints u and v.*

Proof: Suppose u and v are on a cycle. Follow the cycle in some direction from u to v to obtain one path. Then follow the cycle in the opposite direction from u to v to obtain another. Since a cycle has no repeated vertices, the only vertices that lie in both paths are u and v. On the other hand, a path from u to v followed by a path from v to u is a cycle if the paths have no vertices in common other than u and v. □

Definition 5.8 Connected graph *Let $G = (V, E, \varphi)$ be a graph. If for any two distinct elements u and v of V there is a path P from u to v then G is a* **connected graph**. *If $|V| = 1$, then G is connected.*

We make two observations about the definition.

- Because of Theorem 5.2, we can replace "path" in the definition by "walk" or "trail" if we wish.
- The last sentence in the definition is not really needed. To see this, suppose $|V| = 1$. Now G is connected if, for any two *distinct* elements u and v of V, there is a path from u to v. This is trivially satisfied since we cannot find two distinct elements in the one element set V.

The graph of Figure 5.1 (p. 122) is not connected. (There is no path from EN to TM, for example.) The subgraph of this graph induced by the edges {{SH, EN}, {EN, RL}, {EN, CS}} is a connected graph with no cycles. Notice in Figure 5.1, that the relation defined on pairs of vertices u, v by"there exists a path from u to v" partitions the vertices into two subsets: $V_1 = \{$EN, SH, RL, CS$\}$ and $V_2 = \{$MN, SM, SE, TM$\}$. Any two vertices in V_1 can be joined by a path and the same is true for any two vertices in V_2. There is no path connecting a vertex in V_1 to a vertex in V_2.

This is the case in general for a graph $G = (V, E, \varphi)$: The vertex set is partitioned into subsets $V_1, V_2, \ldots, V_m$ such that if u and v are in the same subset then there is a path from u to v and if they are in different subsets there is no such path. The subgraphs $G_1 = (V_1, E_1, \varphi_1), \ldots, G_m = (V_m, E_m, \varphi_m)$ induced by the sets $V_1, \ldots, V_m$ are called the *connected components* of G. Every edge of G appears in one of the connected components. To see this, suppose that $\{u, v\}$ is an edge and

note that the edge is a path from u to v and so u and v are in the same induced subgraph, G_i. By the definition of induced subgraph, $\{u, v\}$ is in G_i.

*Example 5.7 Connected components as an equivalence relation If you've read Section 2, you may have realized that the definition of connected components is a bit sloppy: We need to know that the partitioning into such subsets can actually occur. To see that this is not trivially obvious, define two integers to be "connected" if they have a common factor. Thus 2 and 6 are connected and 3 and 6 are connected, but 2 and 3 are not connected and so we cannot partition the set $V = \{2, 3, 6\}$ into "connected components". We must use some property of the definition of graphs and paths to show that the partitioning of vertices is possible. One way to do this is to construct an equivalence relation.

For $u, v \in V$, write $u \sim v$ if and only if either $u = v$ or there is a walk from u to v. We will use Theorem 5.1 (p. 127) to prove that this is an equivalence relation. It is clear that $\sim$ is reflexive and symmetric. We now prove that it is transitive. Let $u \sim v \sim w$. The walk from u to v followed by the walk from v to w is a walk from u to w. This completes the proof that $u \sim v$ is an equivalence relation. The relation partitions V into subsets $V_1, \ldots, V_m$. □

Exercises

5.3.1. Let C be the set of courses at the university and S the set of students. Let $V = C \cup S$ and let $\{s, c\} \in E$ if and only if student s is enrolled in course c.

(a) Prove that $G = (V, E)$ is a simple graph.

(b) Prove that every cycle of G has an even number of edges.

5.3.2. A graph $G = (V, E)$ is called *bipartite* if V can be partitioned into two sets A and B such that each edge has one vertex in A and one vertex in B. (A partition means that $A \cup B = V$ and $A \cap B = \emptyset$.)

(a) Prove that the example in Exercise 5.3.1 is a bipartite graph.

(b) Prove that every cycle in a bipartite graph has even length.

(c) Suppose that G is a connected bipartite graph. Develop an algorithm to partition the vertices of G into sets A and B such that each edge has one vertex in A and one vertex in B. Prove that your algorithm is correct.

(d) Extend your algorithm to all bipartite graphs.

(e) Prove that the number of ways to choose A and B in a bipartite graph with k connected components is 2^k.

*(f) Prove that a graph is bipartite if and only if every cycle in the graph has even length.

*5.3.3. A *cut edge* or *isthmus* of a connected graph $G = (V, E, \varphi)$ is an edge e such that the removal of e from G leaves a graph which is not connected. A *cut vertex* or *articulation point* of G is a vertex v such that the subgraph induced by $V - \{v\}$ is not connected. For example, in Figure 5.2, edge g is an isthmus and vertex B is a cut vertex. For this problem, assume that G is simple.

(a) If e is a cut edge of G and $v \in \varphi(e)$ is also on another edge $f \neq e$, prove that v is a cut vertex of G.

(b) Give an example of a connected graph that has a cut vertex but does not have an isthmus.

(c) Prove that an edge e of G is a cut edge if and only if it does not lie on a cycle.
Hint. Look for a path that does not contain e but connects the two vertices in $\varphi(e)$.

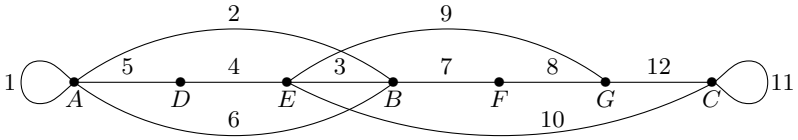
(d) (*Challenge*) Formulate and prove a result similar to the previous one for cut vertices.

5.3.4. A *circuit* (or "closed trail") in a graph $G = (V, E, \varphi)$ is defined exactly as is a cycle in Definition 5.7 except that the "path with vertex sequence $a_1, \ldots, a_n$" is replaced by a "trail with vertex sequence $a_1, \ldots, a_n$." In the next section, we'll define a tree as a connected graph without cycles. Suppose that in this definition and in Definition 5.8 "path"is replaced by "trail" and "cycle" is replaced by "circuit." Would the new definitions of *tree* and *connected graph* describe the same structures as the old definition? Explain.

5.3.5. We are going to describe a process for constructing a graph $G = (V, E, \varphi)$ (with loops allowed). Start with $V = \{v_1\}$ consisting of a single vertex and with $E = \emptyset$. Add an edge e_1, with $\varphi(e_1) = \{v_1, v_2\}$, to E. If $v_1 = v_2$, we have a graph with one vertex and one edge, else we have a graph with two vertices and one edge. Keep track of the vertices and edges in the order added. Here (v_1, v_2) is the sequence of vertices in the order added and the (e_1) is the sequence of edges in order added. Suppose we continue this process to construct a sequence of vertices (not necessarily distinct) added and sequence of *distinct* edges added. At the point where k distinct edges have been added, if v is the last vertex added, then we add a new edge e_{k+1}, different from all previous edges, with $\varphi(e_{k+1}) = \{v, v'\}$ where either v' is a vertex already added or a new vertex. Here is a picture of this process carried out with the edges numbered in the order added, where the vertex sequence is

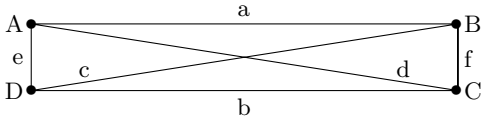
$$(A, A, B, E, D, A, B, F, G, E, C, C, G)$$

$P'(Q)$:

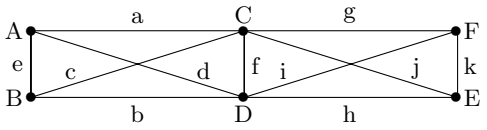


Such a graph is called *Eulerian* or a "graph with an Eulerian trail." By construction, if G is a graph with an Eulerian trail, then there is a trail in G that includes every edge in G. If there is a circuit in G that includes every edge of G then G is called an Eulerian circuit graph or graph with an Eulerian circuit. Thinking about the above example, if a graph has an Eulerian trail but no Eulerian circuit, then all vertices of the graph have even degree except the start vertex and end vertex of the Eulerian trail (they have odd degree). If a graph has an Eulerian circuit then all vertices have even degree. The converses in each case are also true (but take a little work to show). In each of the following graphs, find the longest trail (most edges) and longest circuit. If the graph has an Eulerian circuit or trail, say so.

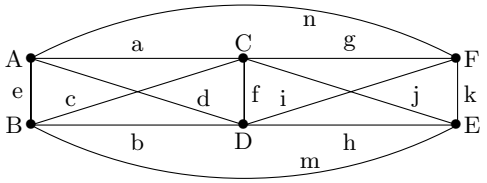
(a)



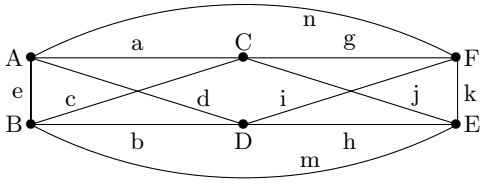
(b)



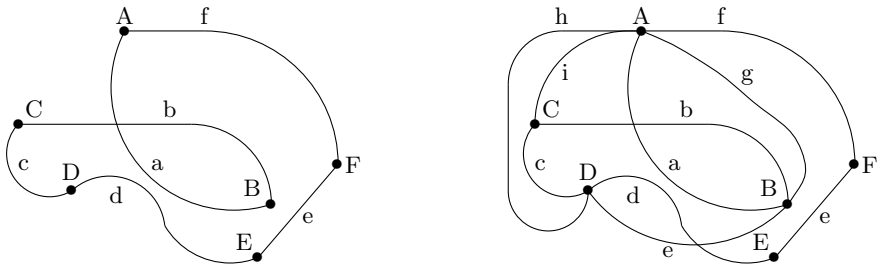
(c)



(d)

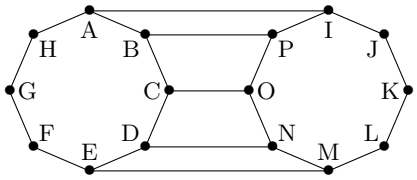


5.3.6. Suppose we start with a graph $G' = (V, E', \varphi')$ that is a cycle and then add additional edges, without adding any new vertices, to obtain a graph $G = (V, E, \varphi)$. As an example, consider

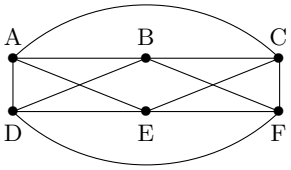


where the first graph $G' = (V, E', \varphi')$ is the cycle induced by the edges $\{a, b, c, d, e, f\}$. A graph that can be constructed from such a two-step process is called a *Hamiltonian* graph. The cycle G' is called a *Hamiltonian* cycle. Alternatively, a cycle in a graph $G = (V, E, \varphi)$ is a Hamiltonian cycle for G if every element of V is a vertex of the cycle. A graph $G = (V, E, \varphi)$ is Hamiltonian if it has a subgraph that is a Hamiltonian cycle for G. For each of the following graphs $G = (V, E, \varphi)$, find a cycle in G of maximum length. State whether or not the graph is Hamiltonian.

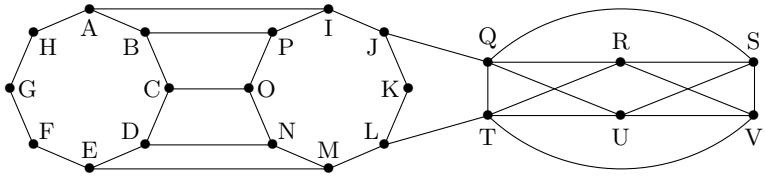
(a)



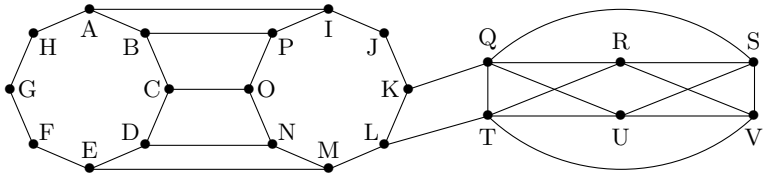
(b)



(c)



(d)



5.4 Trees

Trees play an important role in a variety of algorithms. We have already met decision trees in Chapter 3. In this section, we define trees precisely and look at some of their properties. We study trees further in Section 6.1 and Chapter 9.

Definition 5.9 (Free) Tree *If G is a connected graph without any cycles then G is called a* **tree**. *(If $|V| = 1$, then G is connected and hence is a tree.) A tree is also called a* **free tree**.

The graph of Figure 5.2 (p. 123) is connected but is not a tree. The subgraph of this graph induced by the edges $\{a, e, g\}$ is a tree. If G is a tree, then φ is an injection since if $e_1 \neq e_2$ and $\varphi(e_1) = \varphi(e_2)$, then $\{e_1, e_2\}$ induces a cycle. Because of this, we can think of a tree as a simple graph when we are not interested in names of the edges.

It's natural to ask how many trees can be formed using an n-set V for the vertices. In Example 5.10 (p. 143), we'll prove that the answer is n^{n-2}. Another proof is given in Exercise 5.4.12.

Since the notion of a tree is so important, it will be useful to have some equivalent definitions of a tree. We state them as a theorem

Theorem 5.4 Definitions of tree *If G is a connected graph, the following are equivalent.*

(a) *G is a tree.*

(b) *G has no cycles.*

(c) *For every pair of vertices $u \neq v$ in G, there is exactly one path from u to v.*

(d) *Removing any edge from G gives a graph which is not connected.*

(e) *The number of vertices of G is one more than the number of edges of G.*

Proof: By the definition of a tree, (a) and (b) are equivalent.

Theorem 5.3 can be used to prove that (b) and (c) are equivalent. We leave that as an exercise.

If $\{u, v\}$ is an edge, it follows from (c) that the edge is the only path from u to v and so removing it disconnects the graph. Hence (c) implies (d). We leave it as an exercise to prove that (d) implies (b). This shows that (a), (b), (c), and (d) are all equivalent.

All that remains is (e).

We first show that (b) implies (e). We will use induction on the number of vertices of G. If G has one vertex, it has no edges and so we are done. Otherwise, we claim that G has a vertex u of degree 1; that is, it lies on only one edge $\{u, w\}$. We prove this claim shortly. Remove u and $\{u, v\}$ to obtain a graph H with one less edge and one less vertex. Since G is connected and has no cycles, the same is true of H. Since H has fewer vertices than G, the induction hypothesis tells us that (e) is true for H: there is one more vertex than edge in H. Since H was obtained from G by removing one edge and one vertex, (e) is true for G. It remains to prove the existence of u. Suppose no such u exists; that is, suppose that each vertex lies on at least two edges. We will derive a contradiction. Start at any vertex v_1 of G leave v_1 by some edge e_1 to reach another vertex v_2. Leave v_2 by some edge e_2 different from the edge used to reach v_2. Continue with this process. Since each vertex lies on at least two edges, the process never stops. Hence we eventually repeat a vertex, say

$$v_1, e_1, v_2, \ldots, v_k, e_k, \ldots, v_n, e_n, v_{n+1} = v_k.$$

The edges $e_k, \ldots, e_n$ form a cycle, which is a contradiction.

Now suppose G is a connected graph which is not a tree. It suffices to prove that G has at least as many edges as it has vertices. Why? If we do so, we will have shown

$$\text{((a) is false)} \quad \text{implies} \quad \text{((e) is false)}$$

and hence the contrapositive ((e) is true) implies ((a) is true). On with the proof! By (d) we can remove an edge from G to get a new graph which is still connected. If this is not a tree, repeat the process and keep doing so until we reach a tree. Since (a) implies (b) and (b) implies (e), the number of vertices is now one more than the number of edges. Since we removed edges from G but did not remove vertices, G must have at least as many edges as vertices. □

Example 5.8 Symmetry in graphs and trees Let $G = (V, E)$ be a simple graph. Suppose $\nu : V \to V$ is a isomorphism of G to G. Graph isomorphism is defined in Definition 5.4 (p. 128). We call an isomorphism from something to itself an "endomorphism" or a "symmetry." How much can a symmetry move the vertices of a graph?

It turns out that most graphs have only the trivial endomorphism $\nu(v) = v$ for all $v \in V$, so the vertices can't move at all. On the other hand, if the graph has no edges ($E = \emptyset$) or all possible edges ($E = \mathcal{P}_2(V)$), then every permutation of the vertices in V is a symmetry since the condition $\{u, v\} \in E$ if and only if $\{\nu(u), \nu(v)\} \in E$ in Definition 5.4 is easily seen to hold. What about graphs that are encountered in practice?

The most common graphs in computer science are trees. Most trees have symmetries. For example, suppose $\{u, v\}$, $\{u, w\} \in E$ and v and w are leaves in a tree $T = (V, E)$. (Note that V may have many more vertices besides u, v and w.) We leave it for you to verify that

$$\nu(x) = \begin{cases} w, & \text{if } x = v, \\ v, & \text{if } x = w, \\ x, & \text{otherwise,} \end{cases}$$

is an isomorphism of T. Only the vertices v and w moved. While all vertices might move in an isomorphism of a tree, there are always some that either don't move or don't move "far." We'll prove

Theorem 5.5 *If ν is an isomorphism of the tree $T = (V, E)$ then either*

(a) *there is a vertex v with $\nu(v) = v$ or*

(b) *there is an edge $\{u, v\}$ with $\nu(u) = v$ and $\nu(v) = u$.*

In other words, either a vertex or an edge does not move.

Suppose (a) is not true. We'll define a map $f : V \to E$ and use the Pigeonhole Principle, Theorem 2.5 (p. 55).

If $x \in V$, $\nu(x) \neq x$. Since T is a tree, there is a unique path from x to $\nu(x)$. Let $f(x)$ be the first edge on the path. Since $|E| = |V| - 1$, the Pigeonhole Principle tells us that there must be two vertices v_1 and v_2 with $f(v_1) = f(v_2)$. Thus v_1 and v_2 are the ends of an edge $e = \{v_1, v_2\}$. Since ν is an isomorphism, $\nu(e) = \{\nu(v_1), \nu(v_2)\}$ is also an edge.

We claim $\nu(e) = e$. Draw a picture and try to understand why this is so. (Remember that paths between tree vertices are unique.)

* * * Stop and think about this! * * *

If $\nu(e) \neq e$, here is a way to get from v_1 to v_2 without using e. There is a path P_i from v_i to $\nu(v_i)$ that starts with the edge e. Start at v_1 and follow the part of P_2 after e to $\nu(v_1)$. Traverse the edge $\nu(e) = \{\nu(v_1), \nu(v_2)\}$ from $\nu(v_1)$ to $\nu(v_2)$. Since $e = \{v_1, v_2\}$ is the first edge on P_2, you can follow P_1 backwards until you reach v_2. You have just walked from v_1 to v_2 without using e. Thus there is a path from v_1 to v_2 that does not use e. Since there can be only one path from v_1 to v_2, namely e, our assumption that $\nu(e) \neq e$ must be wrong. Defining $u = v_1$ and $v = v_2$ completes the proof. □

The decision trees in Chapter 3 have some special properties. First, they have a starting point. Second, the edges (decisions) out of each vertex are ordered. We now formalize these concepts.

Definition 5.10 Rooted graph *A pair (G, v), consisting of a graph G and a specified vertex v, is called a* **rooted graph** *with* **root** *v.*

A rooted tree is sometimes simply referred to as a tree, but we will never do so.

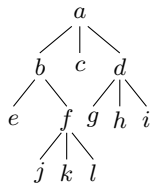


Figure 5.5 A rooted plane tree with root a at the top, as usual. Linear ordering of siblings is left to right.

Definition 5.11 Parent, child, sibling and leaf *Let (T, r) be a rooted tree. If w is any vertex other than r, let $r = v_0, v_1, \ldots, v_k, v_{k+1} = w$, be the unique path from r to w. We call v_k the* **parent** *of w and call w a* **child** *of v_k. Vertices with the same parent are* **siblings**. *A vertex with no children is a* **leaf**.

Definition 5.12 Rooted plane tree *Let (T, r) be a rooted tree. For each vertex, order the children of the vertex. The result is a* **rooted plane tree**, *which we abbreviate to* **RP-tree**. *An RP-tree is also called a* **tree**. *Parents and children are also called* **fathers** *and* **sons**.

Figure 5.5 shows an RP-tree. The sons of b are e and f, the parent of g is d, vertex k has no children, and the siblings of h are g and i. The decision trees of Chapter 3 are RP-trees. When we draw a tree as in Figure 5.5, the root is normally the topmost vertex and all edges are directed downward. In addition, siblings are drawn from left to right following their ordering. For example, the ordering of the siblings $\{j, l, k\}$ is j, then k and then l.

It's clear where "rooted" comes from in RP-tree, but where does "plane" come from? When such a tree is drawn on a piece of paper (a plane), we can start with the root, list its children below it in order, and so on—just like the picture of a decision tree. On the other hand, any rooted tree drawn in the plane has a natural ordering for the children of a vertex v provided v is not the root: Starting at the edge from the parent of v, walk counterclockwise around v, listing the children of v in the order in which their edges are met.

> *We now have two different concepts that are referred to as trees: free trees (Definition 5.9) and RP-trees. How will we keep them straight? When we believe there is no chance of confusion, we may call them trees; otherwise will we call them free trees and RP-trees. Sometimes the distinction is not needed. For example, by Theorem 5.4 the statement that a tree has no cycles applies equally well to RP-trees and free trees since every RP-tree is simply a free tree with a root and orderings.*

In Chapter 9 we'll discuss some recursive aspects of RP-trees including tree *traversal* and *grammars*.

Exercises

5.4.1. Complete the proof of Theorem 5.4.

5.4.2. Let $G = (V, E, \varphi)$ be a connected graph. Using Theorem 5.4 and its proof, do the following.

(a) Prove that G has a cycle if and only if there is some edge e such that the subgraph of G with vertices V and edges $E - \{e\}$ is connected.

(b) Prove that there is a subgraph of G which is a tree and has vertex set V. Such a tree is called a *spanning tree*.

(c) Prove that $|V| \leq |E| + 1$ with equality if and only if G is a tree.

5.4.3. Let $T = (V, E)$ be a tree and let $d(v)$ be the degree of a vertex as defined in Exercise 5.1.1.

(a) Prove that $\sum_{v \in V}(2 - d(v)) = 2$.
Hint. See Exercise 5.1.1.

(b) Prove that, if T has a vertex of degree $m \geq 2$, then it has at least m vertices of degree 1. Vertices of degree 1 are called *leaves* or *terminal vertices*.

(c) Give an example for all $m \geq 2$ of a tree with a vertex of degree m and only m leaves.

(d) Suppose that T has at most one vertex of degree 2. Prove that over half the vertices of T are leaves.

(e) Give an example for all $m > 0$ of a tree with m leaves, $m - 1$ other vertices and at most one vertex of degree 2.

5.4.4. In this exercise, we study how counting edges and vertices in a graph can establish that cycles exist.

(a) Using induction on n, prove:
If $n \geq 0$, a connected graph with v vertices and $v + n$ edges has at least $n + 1$ cycles.

(b) Prove that a graph with v vertices, e edges and c components has at least $c + e - v$ cycles.
Hint. Use (a) for each component.

(c) Show that (a) is best possible, even for simple graphs. In other words, for each n construct a simple graph that has n more edges than vertices but has only $n + 1$ cycles.

5.4.5. Prove that every tree with at least 3 vertices has a cut vertex and a cut edge. (The terms cut edge and cut vertex are defined in Exercise 5.3.3.)

5.4.6. Give an example of a graph that satisfies the specified condition or show that no such graph exists.

(a) A tree with six vertices and six edges

(b) A tree with three or more vertices, two vertices of degree one and all the other vertices with degree three or more.

(c) A disconnected simple graph with 10 vertices, 8 edges and a cycle.

(d) A disconnected simple graph with 12 vertices, 11 edges and no cycles.

(e) A tree with 6 vertices and the sum of the degrees of all vertices 12.

(f) A connected simple graph with 6 edges, 4 vertices, and exactly 2 cycles.

(g) A simple graph with 6 vertices, 6 edges and no cycles.

5.4.7. The *height* of a rooted tree is the the length of the longest path from a leaf of the tree to the root of the tree. A rooted tree in which each non-leaf vertex has at most two children is called a *binary tree*. If each non-leaf vertex has exactly two children, the tree is called a *full binary tree*.

(a) Show that if a binary tree has l leaves and height h then $l \leq 2^h$, or, equivalently, $\log_2(l) \leq h$.

(b) Given that a binary tree has l leaves, what can you say about the maximum value of h?

(c) Given a full binary tree with l leaves, what is the maximum height h?

(d) Given a full binary tree with l leaves, what is the minimum height h?

(e) Given a binary tree of l leaves, what is the minimal height h?

5.4.8. Prove that a full binary tree with n leaves has a total of $2n - 1$ vertices. (The concept of a full binary tree was defined in Exercise 5.4.7.)
Challenge. How many different proofs can you find?

5.4.9. In each of the following cases, state whether or not such a tree is possible.

(a) A binary tree with 35 leaves and height 100.

(b) A full binary tree with 21 leaves and height 21.

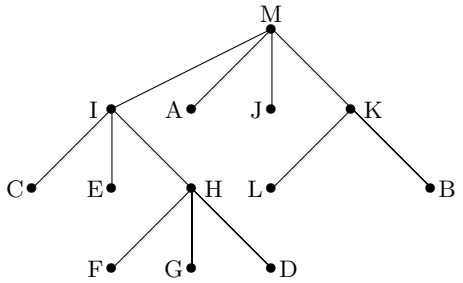
(c) A binary tree with 33 leaves and height 5.

(d) A full binary tree with 65 leaves and height 6.

5.4.10. What is the maximal number of *vertices* in a rooted tree of height h if every vertex has at most k children. What is the maximal number of *leaves* in a rooted tree of height h if every vertex has at most k children?

5.4.11. We are going to define certain important lists of vertices associated with the rooted plane tree, Figure 5.5. These lists can, in a similar fashion, be associated with any rooted plane tree. The first list is $abcdefghijkl$. This list, called the *breadth-first vertex list*, is obtained by starting at the root and reading the vertices of the tree, left to right, as one would read a book. For this definition to work on any tree, you must imagine the tree drawn with the root at the top and all vertices distance one from the root drawn at the next level down, all of distance two at the next level, etc. The second important list is called the *depth-first vertex list*. The depth-first vertex list for Figure 5.5 is $abebfjjfkflfbacadgdhdida$. Note that each vertex appears in the depth-first vertex list a number of times equal to one plus the number of children of that vertex. If one extracts the sublist of first occurrences of each vertex in the depth first list, one gets the list $abefjklcdghi$. This list is called the *pre-order vertex list*. If one extracts the sublist of last occurrences of each vertex in the depth first list, one gets the list $ejklfbcghida$ This list is called the *post-order vertex list*.

(a) For the following rooted plane tree, list the breadth-first, depth-first, pre-order, and post-order vertex lists.



(b) Given that the following is the depth-first vertex list of a rooted plane tree, reconstruct the tree: $MKBKLKMIHDHGHFHIEICIM$.

(c) Is it possible to reconstruct a rooted plane tree given just its pre-order vertex list?

(d) Is it possible to reconstruct a rooted plane tree given its pre-order vertex list and its post-order vertex list?

*5.4.12. Construct a bijection between functions from $\underline{n-2}$ to $\underline{n}$ and trees with $V = \underline{n}$ as follows. Repeatedly remove the leaf with the largest label from the tree until only a two vertex tree remains. When a leaf is removed, list the vertex that it was attached to. This is called the *Prüfer sequence* for the tree. To establish the bijection, you must prove that any list of length $n-2$ chosen from $\underline{n}$ with repetition allowed corresponds is the Prüfer sequence of a tree.
Hint. Show that the largest number *not* in the Prüfer sequence is the vertex that was first removed.

5.5 Directed Graphs (Digraphs)

In the next two sections, we take a look at two important modifications of the concept of a graph.

Look again at Figure 5.2 (p. 123). Imagine now that the symbols a, b, c, d, e, f and g, instead of standing for route names, stand for commodities (applesauce, bread, computers, etc.) that are produced in one town and shipped to another town. In order to get a picture of the flow of commodities, we need to know the directions in which they are shipped. This information is provided by Figure 5.6.

In set theoretic terms, the information needed to construct Figure 5.6 can be specified by giving a pair $D = (V, \varphi)$ where φ is a function with domain $E = \{a, b, c, d, e, f, g\}$ and range $V \times V$. Specifically,

$$\varphi = \begin{pmatrix} a & b & c & d & e & f & g \\ (B,A) & (A,B) & (C,A) & (C,B) & (B,C) & (C,B) & (D,B) \end{pmatrix}.$$

The structure given in Figure 5.6 is an example of a directed graph:

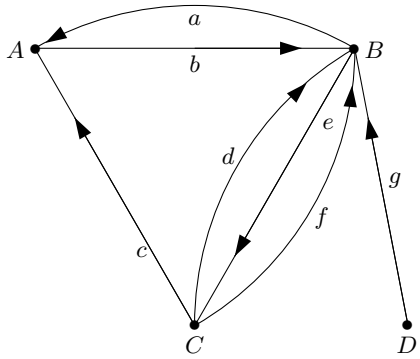


Figure 5.6 Cities with flow of commodities shown.

Definition 5.13 Directed graph *A* **directed graph** *(or* **digraph***) is a triple* $D = (V, E, \varphi)$ *where* V *and* E *are finite sets and* φ *is a function with domain* E *and range* $V \times V$*. We call* E *the set of* **edges** *of the digraph* D *and call* V *the set of* **vertices** *of* D*.*

Note that it is possible that $f(x) = (v, v)$ for $v \in V$. Such an edge x is called a *loop*. A *simple digraph*, like a simple graph, is a pair (V, E) where $E \subseteq V \times V$. Subgraphs, paths, walks, trails and cycles in directed graphs are analogous to the corresponding structures in a graph. A directed graph $D' = (V', E', \varphi')$ is a *directed subgraph* of $D = (V, E, \varphi)$ if $V' \subseteq V$, $E' \subseteq E$ and φ' is the restriction of φ to E' with range $V' \times V'$. A *directed path* in the digraph $D = (V, E, \varphi)$ is a sequence of edges $e_1, \ldots, e_{n-1}$ for which there is a sequence of distinct vertices $a_1, \ldots, a_n$ such that $\varphi(e_i) = (a_i, a_{i+1})$ for $i = 1, 2, \ldots, n-1$. The subdigraph induced by a set of edges $E' \subseteq E$ or the set of vertices $V' \subseteq V$ is defined in a way analogous to the corresponding concept for graphs. Let $D = (V, E, \varphi)$ be a directed graph and let $e_1, \ldots, e_{n-1}$ with vertex sequence $a_1, \ldots, a_n$ be a directed path. If x is an edge of D such that $\varphi(x) = (a_n, a_1)$, then the subgraph induced by the edges $\{e_1, \ldots, e_{n-1}, x\}$ is called a *directed cycle* in D. For example, the subgraph induced by the edges $\{c, b, e\}$ is a directed cycle in the digraph of Figure 5.6. The notions of *edge labeling* and *vertex labeling* extend directly to digraphs. The ideas of connected components and trees are more complicated in digraphs.

Example 5.9 Digraphs and binary relations Simple digraphs appear in mathematics under another important guise: *binary relations*. A binary relation on a set V is simply a subset of $V \times V$. Often the name of the relation and the subset are the same. Thus we speak of the binary relation $E \subseteq V \times V$. If you have absorbed all the terminology, you should be able to see immediately that (V, E) is a simple digraph and that any simple digraph (V', E') corresponds to a binary relation $E' \subseteq V' \times V'$.

What about simple graphs? We can identify $\{u, v\} \in \mathcal{P}_2(V)$ with $(u, v) \in V \times V$ and with $(v, u) \in V \times V$. A binary relation R is called "symmetric" if $(u, v) \in R$ implies $(v, u) \in R$. Thus it seems that simple graphs correspond to symmetric binary relations. This is not quite true since (u, u) would correspond to a loop. We must either allow loops or only look at symmetric binary relations that do not contain (u, u).

An equivalence relation on a set S is also a binary relation $R \subseteq S \times S$: We have $(x, y) \in R$ if and only if x and y are equivalent. Note that this is a symmetric relationship. Which simple graphs (with loops allowed) correspond to equivalence relations? There is a simple description of them. We'll let you look for it.

Functions from a set to itself are another special case of binary relations. Figure 5.7 shows a function and its associated digraph, called a *functional digraph*. Notice that some of the vertices form cycles and the remaining vertices form trees that are attached to the cycles, each tree being

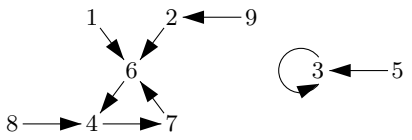


Figure 5.7 The functional digraph associated with the function $\varphi = \left(\begin{smallmatrix} 1&2&3&4&5&6&7&8&9 \\ 6&6&3&7&3&4&6&4&2 \end{smallmatrix}\right)$.

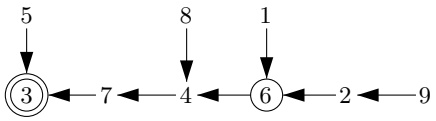


Figure 5.8 The doubly marked digraph built from the functional digraph of Figure 5.7. Removing the arrows gives a doubly marked tree.

attached by one of the vertices on a cycle. For example, the vertices 1, 2, 6, and 9 form a tree that is attached to a cycle by the vertex 6.

One can easily read powers (using composition, not multiplication) of a function from the functional digraph: The function itself comes from directed paths of length 1 and φ^k comes from directed paths of length k.

Permutations are a special case of functions. You should be able to see that, in this case, the functional digraph consists of cycles with no trees attached. □

Example 5.10 The number of labeled trees Let t_n be the number of trees with vertex set $\underline{n}$. It's not hard to draw the possible trees for small values of n. If you do this, you should discover that $t_1 = 1$, $t_2 = 1$, $t_3 = 3$ and $t_4 = 16$. What is the pattern?

It turns out that $t_n = n^{n-2}$. You might try to check this for $t_5 = 125$. How can we prove this formula?

When we know the answer to a problem, we can often use some backwards reasoning or "answer analysis" to figure out how we might solve the problem. Since n^{n-2} is the number of functions from $\underline{n-2}$ to $\underline{n}$, we might try to find a bijection between such functions and trees. (This is done in Exercise 5.4.12.) Unfortunately, it is not at all clear how to proceed with this idea.

It would be much nicer if we looked at functions from $\underline{n}$ to $\underline{n}$ because these lead to functional digraphs: with the function $f \in \underline{n}^{\underline{n}}$, we associate the functional digraph (V, E) where $V = \underline{n}$ and $E = \{(x, f(x)) \mid x \in \underline{n}\}$. Let's pursue this and see if we can generate some ideas.

Look at the functional digraph in Figure 5.7. If we could somehow get rid of the cycles, we would have trees!

Some inspiration is needed. Here it is. The vertices on the cycles are a permutation drawn in cyclic form. For example, in Figure 5.7, the permutation is $(3)(4, 7, 6)$. Maybe we can draw the permutation in some other fashion. We can also write permutations in two line or one line form:

$$\left(\begin{smallmatrix} 3&4&6&7 \\ 3&7&4&6 \end{smallmatrix}\right) \qquad 3, 7, 4, 6.$$

We could simply take the one line form and construct a directed graph from it:

We could also drag along the other vertices that form trees attached to these cyclic vertices. This is shown in Figure 5.8, where you should ignore the circles around the vertices for the time being. We have constructed a tree!

This has one problem, we can no longer tell which vertices were on the cycles. The circles in Figure 5.8 take care of this, the double circle indicating the start of one line form and the single

circle indicating the end. The unique path from the end to the start gives the one line notation, written in reverse. (The directed path is unique because we have constructed a tree.)

Actually our picture has more information than we need: The direction of the edges is determined by the fact that they are all directed toward the root, which is 3 in our example. Thus we can erase the arrowheads on the edges with no loss of information.

This entire process is reversible—Given any tree on $\underline{n}$ in which one vertex is marked with a double circle and one with a single circle, we can recover a unique function which gives this marked tree. Note that the same vertex may have the double circle and the single circle. This happens when there is only one point of the functional digraph on cycles. How many such doubly marked trees are there? By the bijection, there are n^n. Since we form a tree AND mark a vertex with a double circle AND mark a vertex with a single circle, the number is also $t_n \times n \times n$. This completes the proof. □

Exercises

5.5.1. If $D = (V, \varphi)$ is a loopless directed graph, the associated graph $G(D)$ is obtained by removing the directions of the edges. Instead of this rough geometric description, give a definition in terms of sets and functions.
Hint. Define a function with domain $\{(x, y) \mid x, y \in V \text{ and } x \neq y\}$.

5.5.2. We are interested in the number of simple digraphs with $V = \underline{n}$.

(a) Find the number of them.

(b) Find the number of them with no loops.

(c) In both cases, find the number of them with exactly q edges.

5.5.3. An *oriented simple graph* is a simple graph which has been converted to a digraph by assigning an orientation to each edge. The orientation of $\{u, v\}$ can be thought of as a mapping of it to either (u, v) or (v, u). Give an example of a simple digraph that has no loops but is not an oriented simple graph

5.5.4. We are interested in the oriented simple graphs with $V = \underline{n}$. (They are defined in Exercise 5.5.3.)

(a) Find the number of them.

(b) Find the number of them with exactly q edges.

5.5.5. Define an equivalence relation on digraphs that allows you to introduce the notion of unlabeled digraphs. Prove that you have, in fact, defined an equivalence relation.

5.5.6. A digraph is *strongly connected* if, for every two vertices v and w there is a directed path from v to w. From any digraph D, we can construct a simple graph $S(D)$ on the same set of vertices by letting $\{v, w\}$ be an edge of $S(D)$ if and only if at least one of (u, v) and (v, u) is an edge of D. You should find the first three parts of this exercise easy, if you *understand* the meaning of the various concepts—strongly connected, path, $S(D)$, etc.

(a) Prove that, if D is strongly connected, then $S(D)$ is connected.

(b) Construct an example of a digraph D such that D is not strongly connected but $S(D)$ is connected.

(c) Suppose that V_1, V_2 is a partition of the vertices V of a strongly connected digraph D; that is, $V_1 \neq \emptyset$, $V_2 \neq \emptyset$, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \emptyset$. Prove that in D there is an edge from V_1 to V_2 and an edge from V_2 to V_1. (This means that for some $v_1, w_1 \in V_1$ and $v_2, w_2 \in V_2$, both (v_1, v_2) and (w_2, w_1) are edges of D.) We will call such a partition of V "2-way joined."

(d) Suppose that every partition V_1, V_2 of the vertices of D is 2-way joined. Prove that D is strongly connected.

5.5.7. Suppose that we are given a set of statements, for example (a) through (e) in Theorem 5.4, and that we have proved that some statements imply others. Construct a directed graph D as follows. The statements are the vertices V of D. For statements v and w, (v, w) is an edge of D if and only if we have a proof that statement v implies statement w. Prove the claim: "If D is strongly connected, we have proved enough to show that the statements in V are all equivalent." (See Exercise 5.5.6 for a definition of strongly connected.)

5.5.8. For any subset U of the vertices V of a directed graph D, define $d_{\text{in}}(U)$ to be the number of edges of e of D with $\varphi(e)$ of the form (w, u) where $u \in U$ and $w \notin U$. Define $d_{\text{out}}(U)$ similarly.

(a) For v a single vertex, what is $d_{\text{in}}(\{v\})$ in terms of the picture of D?

(b) Prove that $\sum d_{\text{in}}(\{v\}) = \sum d_{\text{out}}(\{v\})$, where the sums both range over all $v \in V$.

(c) Prove that $\sum_{u \in U} d_{\text{in}}(\{u\})$ equals $d_{\text{in}}(U)$ plus the number of non-loop edges of D that have both of their endpoints in U.

(d) Suppose that $d_{\text{in}}(\{v\}) = d_{\text{out}}(\{v\})$ for all $v \in V$. Prove that $d_{\text{in}}(U) = d_{\text{out}}(U)$ for all $U \subseteq V$.

5.5.9. Use the notation of Exercise 5.5.8. Suppose $d_{\text{in}}(\{v\}) = d_{\text{out}}(\{v\})$ for all $v \in V$. Prove that for every edge e of D there is a directed cycle in D that contains e.

5.5.10. Suppose that $S(D)$ is connected, where $S(D)$ is obtained from D by removing the directions of the edges. Use the notation of Exercise 5.5.8. Suppose $d_{\text{in}}(\{v\}) = d_{\text{out}}(\{v\})$ for all $v \in V$. Prove that there is a directed trail that contains every edge of D. Such a trail is called an *Eulerian trail*.

5.5.11. Let G be a connected simple graph.

(a) Suppose that the edges of G can be directed so that the resulting digraph is strongly connected. Prove that G has no isthmuses.

*(b) Suppose that G has no isthmuses. Prove that the edges of G can be directed so that the resulting graph is strongly connected. (This seems to be quite difficult given the material you have had so far. We will return to it later.)

5.5.12. Let $R \subseteq S \times S$ be a binary relation on S. Suppose that $|S| = n$.

(a) How many reflexive binary relations R are there on S?

(b) How many reflexive and symmetric relations R are there on S?

(c) The relation S is *unreflexive* if for all $x \in S$, $(x, x) \notin R$. How many unreflexive, symmetric binary relations R are there on S?

(d) How many symmetric relations R on S are not reflexive?

5.5.13. A binary relation R on S is an *order relation* if it is reflexive, antisymmetric, and transitive. R is *antisymmetric* if for all $(x, y) \in R$ with $x \neq y$, $(y, x) \notin R$. Given an order relation R, the *covering relation* H of R consists of all $(x, z) \in R$ such that there is no y, $x < y < z$, where $(x, y) \in R$. A pictorial representation of the covering relation as a directed graph is called a "Hasse diagram" of H.

(a) Show that the divides relation on $S = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$ is an order relation. By definition, (x, y) is in the divides relation on S is x is a factor of y. Thus, $(4, 12)$ is in the divides relation. $x|y$ is the usual notation for x is a factor of y.

(b) Draw the directed graph of the covering relation of R.

5.5.14. Let R be a binary relation on S. Let T_R be the smallest transitive relation, $R \subseteq T_R$. By "smallest" we mean that if $R \subseteq N \subset T_R$ then N is not transitive. In other words, there is no proper subset of T_R that contains R and is transitive. Note that if R is already transitive then $T_R = R$. T_R is called the *transitive closure* of R. Let $S = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$ and let

$$H = \{(2, 4), (2, 6), (2, 10), (2, 14), (3, 6), (3, 9), (3, 15), (4, 8), (4, 12), (5, 10), (5, 15), (6, 12), (7, 14)\}.$$

Find the transitive closure of H.

5.5.15. A graph is selected uniformly at random from all q-edge simple graphs with vertex set $\underline{n}$.

(a) What is the probability that the graph we have chosen is a tree? (Your answer will depend on q.)

(b) Show that this probability is bigger than $(2/e)^{n-1}/n$ when $q = n - 1$.

Hint. You may use the fact that, by Stirling's formula, $(n-1)! > \left(\frac{n-1}{e}\right)^{n-1}$. Also note that $\binom{N}{q} < N^q/q!$.

*5.6 Computer Representations of Graphs

What is the best way to represent a graph in a computer? That question is based on the mistaken assumption that there is *one best way.* In fact, there are a variety of ways to represent a graph. We'll briefly discuss two common ones: adjacency lists and matrices. For simplicity, *we will limit our attention to simple graphs and digraphs.*

Let $G = (V, E)$ be a graph. For each $v \in V$, keep a list of those $x \in V$ such that $\{v, x\} \in E$. This is a relatively compact method for storing the structure of G. The actual implementation may be with an array or with a linked list. If $D = (V, E)$ is a simple digraph, then we keep two lists for each $v \in V$; one of those $x \in V$ such that $(x, v) \in E$ and one of those $y \in V$ such that $(v, y) \in E$.

The method of linked lists is usually used with RP-trees. If the number of possible sons a vertex can have is not limited, a variation of this method is frequently used. Here's one such variation. Each vertex v has a list of four vertices. If a vertex needed in the list does not exist, that fact is recorded in the list (e.g., by using zero). The four vertices are:

- the parent f of the vertex v;
- the first child of v in the ordering of the children of v;
- the sibling of v that immediately precedes it in the ordering of the siblings of v;
- the sibling of v that immediately follows it in the ordering of the siblings of v.

Matrices are simply doubly indexed arrays. In the most common representation of a simple graph $G = (\underline{n}, E)$, the matrix $A(G)$ is $n \times n$ and

$$a_{i,j} = \begin{cases} 1 & \text{if } \{i, j\} \in E; \\ 0 & \text{otherwise.} \end{cases}$$

For a simple digraph $D = (\underline{n}, E)$, we make a minor adjustment to define $A(D)$:

$$a_{i,j} = \begin{cases} 1 & \text{if } (i, j) \in E; \\ 0 & \text{otherwise.} \end{cases}$$

This representation can waste a considerable amount of space if the graph has relatively few edges. In any event, only about half of $A(G)$ is needed since $a_{i,j} = a_{j,i}$. The matrix representation is useful for some calculations of numbers related to the structure of the graph. See, for example, Exercise 5.6.3.

Exercises

5.6.1. Suppose that G is a bipartite simple graph. (See Exercise 5.3.2 for a definition.) Prove that the vertices can be numbered 1 through $n = |V|$ such that for some k the matrix $A(G)$ has a $k \times k$ block of zeroes in its upper left corner and an $(n-k) \times (n-k)$ block of zeroes in its lower right corner.

5.6.2. Suppose that G is a simple graph with connected components $G_1, \ldots, G_m$. Number the vertices of G by first numbering those in G_1, then those in G_2 and so on. Suppose that G_i has n_i vertices. Provide a description of $A(G)$ like that in Exercise 5.6.1.

5.6.3. This exercise requires familiarity with matrix multiplication. Let $G = (\underline{n}, E)$ be a simple graph. Let $a_{i,j}^{(k)}$ be the (i,j) element of the matrix $(A(G))^k$. Define a *walk* from i to j like a path, but allow repetitions; i.e., a walk is a sequence $e_1, e_2, \ldots, e_{n-1}$ and a sequence of $v_1, \ldots, v_n$ of vertices such that $v_1 = i$, $v_n = j$ and $\varphi(e_i) = \{v_i, v_{i+1}\}$. The *length* of the walk is $n-1$, the number of edges it contains, with repetitions counted.

(a) Prove that there is a walk of length k from i to j in G if and only if $a_{i,j}^{(k)} \neq 0$.

(b) Suppose that $i \neq j$. If $a_{i,j}^{(k)} \neq 0$ for some $k > 0$, let m be the smallest such k. Prove that there is a path of length m between i and j.

(c) Can you find an analog of the previous result for cycles? Be careful!

(d) Prove that G is connected if and only if $(A(G) + I)^k$ contains no zeroes for all sufficiently large k. Prove that all $k \geq n-1$ are sufficiently large.
Hint. With $B = A(G) + I$ and $A(G)^0 = I$, prove that $b_{i,j}^{(k)} = \sum_{t=0}^{k} \binom{k}{t} a_{i,j}^{(t)}$.

5.6.4. State and prove results like those in the previous exercise for directed simple graphs.

5.6.5. A matrix is called *nilpotent* if all sufficiently high powers of it consist entirely of zeroes. Find necessary and sufficient conditions on a simple digraph D for $A(D)$ to be nilpotent.

Notes and References

Because of the relatively modest mathematical prerequisites and the range of applications of graph theory, introductory texts range from high school level to graduate school level and from very applied to very theoretical. Even within one of these compartments, the breadth of graph theory results in a diversity of viewpoints. This is illustrated by the variety in the references.

The book by Ore and Wilson [10] is a delightful elementary introduction. The books by Berge [1], Bondy and Murty [3], Chartrand and Lesniak [4] and Tutte [12] are more advanced than our text. The remaining books are at a level similar to our text, although they often contain some more advanced topics. If you are particularly interested in applications, see the book by Roberts [11]. If your are interested in algorithms, see the book by McHugh [8].

Other applications of the Pigeonhole Principle to prove properties of trees can be found in [7]. Other proofs of the formula $t_n = n^{n-2}$ for trees can be found in Moon's article [9].

1. Claude Berge, *The Theory of Graphs and Its Applications*, Dover (2001).
2. Béla Bollabás, *Graph Theory: An Introductory Course*, Springer-Verlag (1971).
3. J. Adrian Bondy and U.S.R. Murty, *Graph Theory with Applications*, American Elsevier (1976).
4. Gary Chartrand and Linda Lesniak, *Graphs and Digraphs*, 4th ed., CRC Press (2004).
5. Alan Gibbons, *Algorithmic Graph Theory*, Cambridge Univ. Press (1985).
6. Ronald J. Gould and Ron Gould, *Graph Theory*, Bejamin Cummings (1988).
7. N. Graham, R.C. Entringer and L.A. Székely, New tricks for old trees: Maps and the Pigeonhole Principle, *Amer. Math. Monthly* **101** (1994) 664–667.
8. James A. McHugh, *Algorithmic Graph Theory*, Prentice-Hall (1990).
9. John W. Moon, Various proofs of Cayley's formula for counting trees, in Frank Harary (ed.), *A Seminar on Graph Theory*, Holt, Rinehart and Winston (1967).
10. Oystein Ore and Robin J. Wilson, *Graphs and Their Uses*, Mathematical Assn. of America (1990).
11. Fred S. Roberts, *Applied Combinatorics*, Prentice-Hall (1984).
12. William T. Tutte, *Graph Theory*, Encyclopedia of Mathematics and Its Applications, Vol. 21, Cambridge Univ. Press (1984).

CHAPTER 6

A Sampler of Graph Topics

Introduction

A tree is a very important type of graph. For this reason, we've devoted quite a bit of space to them in this text. In Chapter 3, we used decision trees to study the listing, ranking and unranking of functions and to briefly study backtracking. In the next section, we'll focus on "spanning trees." Various types of spanning trees play important roles in many algorithms; however, we will barely touch these applications.

"Graph coloring" problems have been studied by mathematicians for some time and there are a variety of interesting results, some of which we'll discuss. The subject originated from the problem of coloring the countries on a map so that no adjacent countries have the same color. The subject of map colorings is discussed in the Section 6.3, where we consider planar graphs.

If you attempt to draw a graph on a piece of paper, you will often find that you can't do it unless you allow some edges to cross. There are some graphs which can be drawn in the plane without any edges crossing; e.g., all trees. These are called *planar graphs*. Drawing a graph without edges crossing is called *embedding the graph*. K_5, the five vertex complete graph (all edges present) cannot be embedded in the plane. Try it. Can you *prove* that it can't be embedded? It's not clear how to go about *proving* that you haven't somehow missed a clever way to embed it. The impossiblity of embedding K_5 is one of the things that we will prove in Section 6.2.

Aspects of planar graphs of interest to us are coloring, testing for planarity and circuit design. Our discussion of coloring planar maps relies slightly on Section 6.2 and our discussion of a planarity algorithm relies on Section 6.1.

The edges of a graph can be thought of as pipes that hold a fluid. This leads to the idea of *network flows*. We can also interpret the edges as roads, telephone lines, etc. The central practical problem is to maximize, in some sense, the flow through a network. This has important applications in industry. In Section 6.4, we will discuss the underlying concepts and develop, with some gaps, an algorithm for maximizing flow. The theory of flows in networks has close ties with "linear programming" and "matching theory." We will explore the latter briefly.

We can ask what "typical" large graphs are like. For example,

- How many leaves does a typical tree have?
- How many "triangles"—three vertices all joined by edges—does a typical graph have?

- How small can we make the function $q(n)$ and still have most n-vertex, q-edge simple graphs connected?

Questions like these are discussed in Section 6.5.

Finally, we introduce the subject of "finite state machines" in Section 6.6. They provide a theoretical basis for studying questions related to the computability and complexity of functions and algorithms. Various classes of finite state machines are closely associated with various classes of grammars, an association we'll explore briefly in Section 9.2. In Example 11.2 (p. 310), we'll see how some machines provide a method for solving certain types of enumeration problems.

The sections in this chapter are largely independent of each other. Other parts of the book do not require the material in this chapter. If you are not familiar with $\Theta(\ )$ and $O(\ )$ notation, you will need to read Appendix B for some of the examples in this chapter.

6.1 Spanning Trees

Here's the definition of what we'll be studying in this section.

Definition 6.1 Spanning tree *A* **spanning tree** *of a (simple) graph $G = (V, E)$ is a subgraph $T = (V, E')$ which is a tree and has the same set of vertices as G.*

Example 6.1 Since a tree is connected, a graph with a spanning tree must be connected. On the other hand, you were asked to prove in Exercise 5.4.2 (p. 139) that every connected graph has a spanning tree. Thus we have: A graph is connected if and only if it has a spanning tree. It follows that, if we had an algorithm that was guaranteed to find a spanning tree whenever such a tree exists, then this algorithm could be used to decide if a graph is connected. □

In this section, we study minimum weight spanning trees and lineal spanning trees.

Minimum Weight Spanning Trees

Suppose we wish to install "lines" to link various sites together. A site may be a computer installation, a town or a spy. A line may be a digital communication channel, a rail line or a contact arrangement. We'll assume that

- a line operates in both directions;
- it must be possible to get from any site to any other site using lines;
- each possible line has a cost (rental rate, construction costs or likelihood of detection) independent of each other line's cost;
- we want to choose lines to minimize the total cost.

We can think of the sites as vertices V in a graph, the lines as edges E and the costs as a function λ from the edges to the real numbers. Let $T = (V, E')$ be a subgraph of $G = (V, E)$. Define $\lambda(T)$, the *weight* of T, to be the sum of $\lambda(e)$ over all $e \in E'$. Minimizing total cost means choosing T so that $\lambda(T)$ is a minimum. Getting from one site to another means choosing T so that it is connected. It follows that we should choose T to be a spanning tree—if T had more edges than in a spanning tree, we could delete some; if T had less, it would not be connected. (See Exercise 5.4.2 (p. 139).) We call such a T a *minimum weight spanning tree* of (G, λ), or simply of G, with λ understood from context.

How can we find a minimum weight spanning tree T? One approach is to construct T by adding an edge at a time in a greedy way. Since we want to minimize the weight, "greedy" means keeping the weight of each edge we add as low as possible. Here's such an algorithm.

Theorem 6.1 Prim's Algorithm (Minimum Weight Spanning Tree) *Let $G = (V, E)$ be a simple graph with edge weights given by λ. If the following algorithm stops with $V' \neq V$, G has no spanning tree; otherwise, (V, E') is a minimum weight spanning tree for G.*

1. Start: *Let $E' = \emptyset$ and let $V' = \{v_0\}$ where v_0 is any vertex in V.*
2. Possible Edges: *Let $F \subseteq E$ be those edges $\{x, y\}$ with $x \in V'$ and $y \notin V'$. If $F = \emptyset$, stop.*
3. Choose Edge Greedily: *Let $f = \{x, y\}$ be such that $\lambda(f)$ is a minimum over all $f \in F$. Replace V' with $V' \cup \{y\}$ and E' with $E' \cup \{f\}$. Go to Step 2.*

Proof: We begin with the first part; i.e, if the algorithm stops with $V' \neq V$, then G has no spanning tree. Suppose that $V' \neq V$ and that there is a spanning tree. We will prove that the algorithm does not stop at V'. Choose $u \in V - V'$ and $v \in V'$. Since G is connected, there must be a path from u to v. Each vertex on the path is either in V' or not. Since $u \notin V'$ and $v \in V'$, there must be an edge f on the path with one end in V' and one end not in V'. But then $f \in F$ and so the algorithm does not stop at V'.

We now prove that, if G has a spanning tree, then (V, E') is a minimum weight spanning tree. One way to do this is by induction: We will prove that at each step there is a minimum weight spanning tree of G that contains E'.

The starting case for the induction is the first step in the algorithm; i.e., $E' = \emptyset$. Since G has a spanning tree, it must have a minimum weight spanning tree. The edges of this tree obviously contain the empty set, which is what E' equals at the start.

We now carry out the inductive step of the proof. Let V' and E' be the values going into Step 3 and let $f = \{x, y\}$ be the edge chosen there. By the induction hypothesis, there is a minimum weight spanning tree T of G that contains the edges E'. If it also contains the edge f, we are done. Suppose it does not contain f. We will prove that we can replace an edge in the minimum weight tree with f and still achieve minimum weight.

Since T contains all the vertices of G, it contains x and y and, also, some path P from x to y. Since $x \in V'$ and $y \notin V'$, this path must contain an edge $e = \{u, v\}$ with $u \in V'$ and $v \notin V'$. We now prove that removing e from T and then adding f to T will still give a minimum spanning tree.

By the definition of F in Step 2, $e \in F$ and so, by the definition of f, $\lambda(e) \geq \lambda(f)$. Thus the weight of the tree does not increase. If we show that the result is still a tree, this will complete the proof.

The path P together with the edge f forms a cycle in G. Removing e from P and adding f still allows us to reach every vertex in P and so the altered tree is still connected. It is also still a tree because it contains no cycles—adding f created only one cycle and removing e destroyed it. This completes the proof that the algorithm is correct. □

This proof illustrates an important technique for proving that algorithms are correct:

Make an assertion about the algorithm and then prove it inductively.

In this case the assertion was the existence of a minimum weight spanning tree having certain edges. Induction on the number of those edges started easily and the inductive step was not too difficult.

We could construct an even greedier algorithm: At each time add the lowest weight edge that does not create a cycle. The intermediate graphs (V', E') that are built in this way may not be connected; however, if (V, E) was connected, the end result will be a minimum weight spanning tree. We leave it to you to formulate the algorithm carefully and prove that it works in Exercise 6.1.5. This algorithm, with some tricky, efficient handling of the data structures is called *Kruskal's Algorithm.*

Example 6.2 A more general spanning tree algorithm The discussion so far has centered around choosing edges that will be in our minimum weight spanning tree. We could also choose edges that will *not be in* our minimum weight spanning tree. This can be done by selecting a cycle of edges, none of which have been rejected, and then rejecting an edge for which λ is largest among the edges in the cycle. When no more cycles remain, the remaining edges form a minimum weight spanning tree. We leave it to you to prove this. (Exercise 6.1.1)

These ideas can be combined: We have two sets A and R of edges that begin by both being empty. At each step, we somehow add an edge to either A or R and claim that there exists a minimum weight spanning tree that contains all of the edges in A and none of the edges in R. Of course, this will be true at the start. The proof that it is true in general will use induction and will depend on the specific algorithm used for adding edges to A and R.

What sort of algorithms can be built using this idea? We could of course simply use the greedy algorithm for adding edges to A all the time, or we could use the greedier algorithm for adding edges to A all the time, or we could use the cycle approach mentioned at the start of this example to add edges to R. Something new: we could sometimes add edges to A and sometimes to R, whichever is more convenient. This can be useful if we are finding the tree by hand. □

Example 6.3 How fast is the algorithm? We'll analyze the minimum weight (or cost) spanning tree algorithm. Here's a brief description of it. We let $G = (V, E)$ be the given simple graph.

1. Initialize: Select a vertex $v \in V$ and let the tree T consist of v.
2. Select an edge: If there are no edges between V_T (the vertices of T) and $V - V_T$, stop; otherwise, add the one of minimum cost to T and go to Step 2.

Of course, when we add an edge to T, we also add the vertex on it that was not already in T. When the algorithm stops, V_T is the vertex set of the component of G containing v. If G is connected, T is a minimum cost spanning tree.

Suppose that T currently has k vertices. How much work is required to add the next edge to T in the worst case? If the answer is t_k, then the worst case running time is $O(t_1 + \cdots + t_{|V|-1})$. We can't determine t_k since we didn't specify enough details in Step 2. We'll fill them in now. For each vertex u in T, we look at all edges containing it. If $\{u, x\}$ is such an edge, we check to see if $x \in V_T$ and, if not, we check to see if $\{u, x\}$ is the least cost edge found so far in the execution of Step 2. Both these checks can be performed in constant time; i.e, the time does not depend on $|V_T|$ or the size of G. Since we examine at most $|E|$ edges, $t_k = O(|E|)$. Since k ranges from 1 to $|V| - 1$ for a connected graph G, the worst case running time is $O(|V|\,|E|)$.

We cannot say that the worst case running time is $\Theta(|V|\,|E|)$ because we've said "at most" in our argument.

Are there faster algorithms than this? Assuming no one has organized our data according to edge costs, we must certainly look at each edge cost at least once, so any algorithm must have best case running time at least $\Theta(|E|)$. Algorithms with worst case running times $\Theta(|E| \ln\ln |V|)$ and $\Theta(|V|^2)$ are known. If G has a lot of edges, we could have $|E| = \Theta(|V|^2)$ and so $\Theta(|V|^2)$ is $\Theta(|E|)$.

Does this mean that the $\Theta(|V|^2)$ algorithm is best for a graph with about $|V|^2/4$ edges, the typical number in a "random" graph? Not necessarily. To illustrate why not, *suppose* that the running time of the first algorithm is close to $4|E| \ln\ln |V|$ and that of the second is close to $3|V|^2$. The first algorithm will be better as long as

$$3|V|^2 \;>\; 4|E| \ln\ln |V| \;=\; |V|^2 \ln\ln |V|.$$

Solving for $|V|$, we obtain $|V| > \exp(e^3) \;=\; 5 \times 10^8$. This means that the $\Theta(|V|^2 \ln\ln |V|)$ algorithm, which is slower when $|V|$ is very large, would actually be faster for all practical values of $|V|$. (Remember, this is *hypothetical* because we assumed the values of the constants in the $\Theta(\cdots)$ expressions.) See Example B.4 (p. 373) for further discussion. □

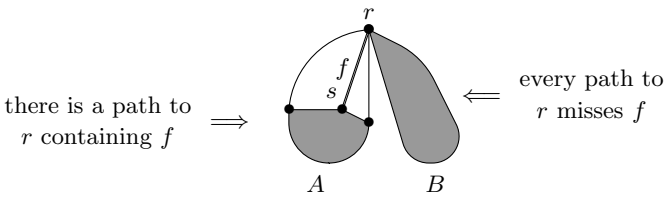


Figure 6.1 An example of the division of vertices for the lineal spanning tree induction. The subgraphs A and B are shaded.

Lineal Spanning Trees

If we simply want to find any spanning tree of G, we can choose any values for the function λ, and use the minimal weight spanning tree algorithm of Theorem 6.1. Put another way, in Step 3 we may choose any edge $f \in F$. Sometimes it is important to restrict the choice of f in some way so that the spanning tree will have some special property other than being minimal.

An important example of such a special property concerns certain rooted spanning trees. To define the trees we are interested in, we borrow some terminology from genealogy.

Definition 6.2 Lineal spanning tree *Let x and y be two vertices in a rooted tree with root r. If x is on the path connecting r to y, we say that y is a* **descendant** *of x. (In particular, all vertices are descendants of r.) If one of u and v is a descendant of the other, we say that $\{u, v\}$ is a* **lineal pair**. *A* **lineal spanning tree** *or* **depth first spanning tree** *of a connected graph $G = (V, E)$ is a rooted spanning tree of G such that each edge $\{u, v\}$ of G is a lineal pair.*

To see some examples of a lineal spanning tree, look back at Figure 5.5 (p. 139). It is the lineal spanning tree of a graph, namely itself. We can add some edges to this graph, for example $\{a, f\}$ and $\{b, j\}$ and still have Figure 5.5 as a lineal spanning tree. On the other hand, if we added the edge $\{e, j\}$, the graph would not have Figure 5.5 as a lineal spanning tree.

How can we find a lineal spanning tree of a graph? That question may be a bit premature—we don't even know when such a tree exists. We'll prove

Theorem 6.2 Lineal spanning tree existence *Every connected graph G has a lineal spanning tree. In fact, given any vertex r of G, there is a lineal spanning tree of G with root r.*

Proof: Our proof will be by induction on the number of vertices in G. The theorem is trivially true for a graph with one vertex. Suppose we know that the claim is true for graphs with less than n vertices. Let $G = (V, E)$ have n vertices and let $f = \{r, s\} \in E$. We will prove that G has a lineal spanning tree with root r.

Let $S \subseteq V$ be those vertices of G that can be reached by a path starting at r and containing the edge f. Note that $r \notin S$ because a path cannot contain repeated vertices; however, $s \in S$. Let $R = V - S$. If $x \in R$, every path from r to x misses s, for if not we could simply go from r to s on f and then follow the path to x.

Let A be the subgraph of G induced by S and let B be the subgraph of G induced by R. (Recall that the subgraph induced by S is the set of all edges in G whose end points both lie in S.) We now prove:

$$\text{Each edge of } G \text{ that does not contain } r \text{ lies in either } A \text{ or } B. \qquad 6.1$$

Suppose we had an edge $\{u, v\}$ with u in A and $v \neq r$ in B. There is a path from r to u using f. By adding v to the path, we conclude that $v \in S$, a contradiction.

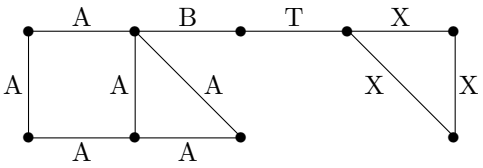
We claim that A is connected and B is connected. Suppose x and y are vertices that are both in A or both in B. Since G is connected, there is a path joining them in G. If an edge of G does not lie in A or in B, then, by (6.1), r is one of its vertices. A path starting in A or B, leaving it and then returning would therefore contain r twice, contradicting the definition of a path. Thus the path lies entirely in A or entirely in B.

Since neither R nor S is empty, each of A and B have less vertices than G. Thus, since A and B are connected, we may apply the induction hypothesis to A and to B. Let $T(A)$ be the lineal spanning tree of A rooted at s and let $T(B)$ be the lineal spanning tree of B rooted at r.

Join $T(A)$ to $T(B)$ by f to produce a connected subgraph T of G with vertex set V and root r. Since $T(A)$ and $T(B)$ have no cycles, it follows that T is a spanning tree of G.

To complete the proof, we must show that T is lineal. Let $e = \{u, v\} \in E$. If one of u and v is r, then e is a lineal pair of T. Suppose that $r \notin e = \{u, v\}$. By (6.1), e lies in A or B. Since $T(A)$ and $T(B)$ are lineal spanning trees, e is a lineal pair of either $T(A)$ or $T(B)$ and hence of T. □

Example 6.4 Bicomponents of graphs Let $G = (V, E)$ be a simple graph For $e, f \in E$ write $e \sim f$ if either $e = f$ or there is a cycle of G that contains both e and f. We claim that this is an equivalence relation. To see what we're talking about, let's look at an example.



6.2

The edges fall into four equivalence classes, which we've arbitrarily called A, B, T and X. Each edge has the letter of its equivalence class next to it. Notice that the vertices *do not* fall into equivalence classes because some of them would have to belong to more than one equivalence class.

Now we'll prove that we have an equivalence relation by using Theorem 5.1 (p. 127). The reflexive and symmetric parts are easy. Suppose that $e \sim f \sim g$. If $e = g$, then $e \sim g$, so suppose that $e \neq g$. Let $e = \{v_1, v_2\}$. Let $C(e, f)$ be the cycle containing e and f and $C(f, g)$ the cycle containing f and g. In $C(e, f)$ there is a path P_1 from v_1 to v_2 that does not contain e. Let x and $y \neq x$ be the first and last vertices on P_1 that lie on the cycle containing f and g. We know that there must be such points because the edge f is on P_1. Let P_2 be the path in $C(e, f)$ from y to x containing e. In $C(f, g)$ there is a path P_3 from x to y containing g. We have shown that P_2 followed by P_3 defines a cycle containing e and g. Hence $e \sim g$.

Since $\sim$ is an equivalence relation on the edges of G, it partitions them. If the partition has only one block, then we say that G is a *biconnected graph*. If E' is a block in the partition, the subgraph of G induced by E' is called a *bicomponent* of G. Note that the bicomponents of G are not necessarily disjoint: Bicomponents may have vertices in common (but *never* edges). The picture (6.2) has four bicomponents.

Finding the bicomponents of a graph is important when we wish to decide if the graph can be drawn in the plane so that no edges cross. We discuss this briefly at the end of Section 6.3.

Biconnectivity is closely related to lineal spanning trees. Suppose T is a lineal spanning tree of G and that the vertices x and y are in the same bicomponent of G. Then either

$$\{x, y\} \text{ is an edge that is a bicomponent by itself}$$

or

$$\text{there is a cycle containing } x \text{ and } y.$$

In either case, since T is a lineal spanning tree, $\{x, y\}$ is a lineal pair. This leads to an algorithm for finding bicomponents: Suppose $e = \{x, y\}$ is an edge that is not in T. If f is an edge on the path

from x to y in T, write $e \sim f$. As it stands, $\sim$ is not an equivalence relation; however, it can be made into one by adding what is needed to insure reflexivity, symmetry and transitivity. In the resulting relation it turns out that, $e \sim f$ if and only if e and f are in the same bicomponent. (This requires proof, which we omit.) You might like to experiment with this idea. □

Exercises

6.1.1. A cycle approach to forming a minimum weight spanning tree was discussed in Example 6.2: Throw away largest weight edges that do not disconnect the graph. Prove that it actually leads to a minimum weight spanning tree as follows.

(a) Let T be a minimum weight spanning tree and let e be the first edge that is removed by the algorithm but is contained in T. Prove that T with e deleted consists of two components, T_1 and T_2.

(b) Call any edge in the original graph that has one end in T_1 and one in T_2 a *connector*. Prove that, if f is a connector, then $\lambda(f) \geq \lambda(e)$.

(c) Let T^* be the spanning tree produced by the algorithm. Prove that, if e is added to T^*, then the resulting graph has a cycle containing e and some connector f.

(d) Let f be the edge in (c). Prove that $\lambda(f) \geq \lambda(e)$.

(e) Let f be the edge in (c). Prove that T with e removed and f added is also a minimum weight spanning tree.

*(f) Complete the proof.

6.1.2. Let G be a connected simple graph and let B_1 and $B_2 \neq B_1$ be two bicomponents of G. Prove that B_1 and B_2 have at most one vertex in common.

6.1.3. Let G be a connected simple graph. Let $Q(G)$ be the set of bicomponents of G and let $P(G)$ be the set of all vertices of G that belong to more than one bicomponent; i.e., $P(G)$ is the union of the sets $H \cap K$ over all pairs $H \neq K$ with $H, K \in Q(G)$. Define a simple bipartite graph $\mathcal{B}(G)$ with vertex set $W = P(G) \cup Q(G)$ and $\{u, X\}$ an edge if $u \in P(G)$ and $u \in X \in Q(G)$. (See Exercise 5.3.2 for a definition of bipartite.)

(a) Construct three examples of $\mathcal{B}(G)$, each containing at least four vertices.

(b) Prove that $\mathcal{B}(G)$ is a connected simple graph.

(c) Prove that $\mathcal{B}(G)$ is a tree.
Hint. Prove that a cycle in $\mathcal{B}(G)$ would lead to a cycle in G that involved edges in different bicomponents.

(d) Prove that $P(G)$ is precisely the articulation points of G. (See Exercise 5.3.3 for a definition.)

6.1.4. Using the proof of Theorem 6.1, prove: If λ is an *injection* from E to $\mathbb{R}$ (the real numbers), then the minimum weight spanning tree is unique.

*6.1.5. We will study the greedier algorithm that was mentioned in the text. Suppose the graph $G = (V, E, \varphi)$ has n vertices. From previous work on trees, we know that any spanning tree has $n - 1$ edges. Let $g_1, g_2, \ldots, g_{n-1}$ be the edges in the order chosen by the greedier algorithm. Let $e_1, e_2, \ldots, e_{n-1}$ be the edges in any spanning tree of G, ordered so that $\lambda(e_i) \leq \lambda(e_{i+1})$. Our goal is to prove that $\lambda(g_i) \leq \lambda(e_i)$ for $1 \leq i < n$. It follows immediately from this that the greedier algorithm produces a minimum weight spanning tree.

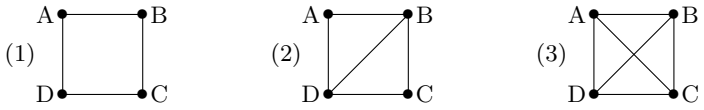
(a) Prove that the vertices of G together with any k edges of G that contain no cycles is a graph with $n - k$ components each of which is a tree.

(b) Let G_k be the graph with vertices V and edges $g_1, \ldots, g_k$. Let H_k be the graph with vertices V and edges $e_1, \ldots, e_k$. Prove that one of the edges of H_{k+1} can be added to G_k to give a graph with no cycles.
Hint. Prove that there is some component of H_{k+1} that contains vertices from more than one component of G_k and then find an appropriate edge in that component of H_{k+1}.

(c) Prove that $\lambda(g_i) \leq \lambda(e_i)$ for $1 \leq i < n$.

6.1.6. Using the result in Exercise 6.1.5, prove that whenever λ is an injection the minimum weight spanning tree is unique.

6.1.7. For each of the following graphs:



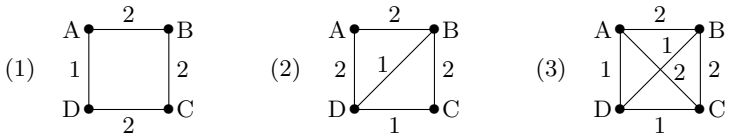
(a) Find all spanning trees.

(b) Find all spanning trees up to isomorphism; that is, find all distinct trees when the vertex labels are removed.

(c) Find all depth-first spanning trees rooted at A.

(d) Find all depth-first spanning trees rooted at B.

6.1.8. For each of the following graphs:



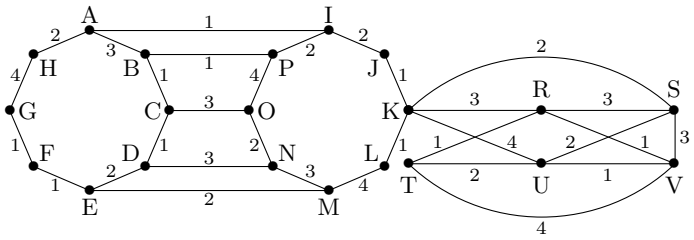
(a) Find all minimal spanning trees.

(b) Find all minimal spanning trees up to isomorphism; that is, find all distinct trees when the vertex labels are removed.

(c) Find all minimal depth-first spanning trees rooted at A.

(d) Find all minimal depth-first spanning trees rooted at B.

6.1.9. In the following graph, the edges are weighted either 1, 2, 3, or 4.



(a) Find a minimal spanning tree using the method of Theorem 6.1.

(b) Find a minimal spanning tree using the method of Example 6.2.

(c) Find a minimal spanning tree using the method of Exercise 6.1.5.

(d) Find a depth-first spanning trees rooted at K.

6.2 Coloring Graphs

Example 6.5 Register allocation Optimizing compilers use a variety of techniques to produce faster code. One obvious way to produce faster code is to keep variables in registers so that memory references are eliminated. Unfortunately, there are often not enough registers available to do this, so choices must be made. For simplicity, assume that the registers and variables are all the same size. Suppose that, by some process, we have gotten a list of variables that we would like to keep in registers.

Can we keep them in registers? If the number of variables does not exceed the number of available registers, we can obviously do it. This sufficient condition is not necessary: We may have two variables that are only used in two separate parts of the program. They could share a register.

This suggests that we can define a binary relation among variables. We could say that two variables are "compatible" if they may share a register. Alternatively, we could say that two variables "conflict" if they cannot share a register. Two variables are either compatible or in conflict, but not both. Thus we can derive one relation from the other and it is rather arbitrary which we focus on. For our purposes, the conflict relation is better.

Construct a simple graph whose vertices are the variables. Two variables are joined by an edge if and only if they conflict. A register assignment can be found if and only if we can find a function λ from the set of vertices to the set of registers such that whenever $\{v, w\}$ is an edge $\lambda(v) \neq \lambda(w)$. (This just says that if v and w conflict they must have different registers assigned to them.) This section studies such ""vertex labelings" λ. □

Definition 6.3 Graph coloring *Let $G = (V, E)$ be a simple graph and C a set. A* **proper coloring** *of G using the "colors" in C is a function $\lambda: V \to C$ such that $\lambda(v) \neq \lambda(w)$ whenever $\{v, w\} \in E$.*

Some people omit "proper" and simply refer to a "coloring." A solution to the register allocation problem is a proper coloring of the "conflict graph" of the variables using the set of registers as colors.

Given G and C, we can ask for reasonably fast algorithms to answer various questions about proper colorings:

1. Does there exist a proper coloring of G with C?
2. What is a good way to find a proper coloring of G with C, if one exists?
3. How many proper colorings of G with C are there?

Question 1 could be answered by an algorithm that attempts to construct a proper coloring and fails only if none exist. It could also be answered by calculating the number of proper colorings and discovering that this number is zero. Before trying to answer these questions, let's look at more examples.

Example 6.6 Scheduling problems Register allocation is an example of a simple *scheduling problem*. In this terminology, variables are scheduled for storage in registers. A scheduling problem can involve a variety of constraints. Some of these can be sequential in nature: Problem definition must occur before algorithm formulation, which must in turn occur before division of programming tasks. Others can be conflict avoidance like register allocation. The simplest sort of conflict avoidance conditions are of the form "v and w cannot be scheduled together," which we encountered with register allocation. These can be phrased as graph coloring problems.

Here's an example. Suppose we want to make up a course schedule that avoids obvious conflicts. We could let the vertices of our graph be the courses. Two courses are connected by an edge if we expect a student in one course will want to enroll in the other during the same term. The colors are the times at which courses meet. □

Example 6.7 Map coloring In loose terms, a *map* is a collection of regions, called countries and water, that partition a sphere. (A sphere is the *surface* of a ball.) To make it easy to distinguish regions that have a common boundary, they should be different colors. ("Common boundary" means an actual line segment or curve and not just a single point.) This can be formulated as a graph coloring problem by letting the regions be vertices and by joining two vertices with an edge if the corresponding regions have a common boundary. What problems, if any, are caused by our loose definition of a map? A country may consist of several pieces, like the United States which includes Alaska and Hawaii. This is ruled out in a careful definition of a map.

It is easy to find a map that requires four colors. Try to find such a map yourself. Later we will prove that any map can be colored with five colors. How many colors are needed? From the past few sentences, at least four and at most five. Four colors suffice. This fact is known as the *Four Color Theorem*. At present, the only way this can be proved is by extensive computer calculations. This was done by Appel and Haken in 1976.

Maps can be defined on more complicated surfaces (like a torus—the surface of a doughnut). For each such surface S there is a number $n(S)$ such that some map requires $n(S)$ colors and no map requires more. A fairly simple formula has been found and proved for $n(S)$. It is somewhat amusing that computer calculations are needed only in what appears to be the simplest case—when S is equivalent to a sphere. □

How can we construct a proper coloring of a graph? Suppose we have n vertices and c colors. We could systematically try all c^n possible assignments of colors to vertices until we find one that works or we find that there is no proper coloring. Backtracking on a decision tree can save considerable time. A decision corresponds to assigning a color to a vertex. Suppose that we are at some node t in the decision tree where we have already colored vertices $v_1, \ldots, v_k$ of the graph. The edges leading out of t correspond to the different ways of coloring v_{k+1} so that it is not the same color as any of $v_1, \ldots, v_k$ that are adjacent to it. It is not clear how fast this algorithm is or if we could find a substantially better one.

We'll abandon the construction problem in favor of the counting problem. We will prove

Theorem 6.3 Chromatic polynomial existence *Let G be a simple graph with n vertices. There is a polynomial $P_G(x)$ of degree n such that for each positive integer m, the number of ways to properly color G using $\underline{m}$ is $P_G(m)$.*

$P_G(x)$ is called the *chromatic polynomial* of G. Various properties of it are found in the exercises.

We'll give two proofs of the theorem. The first is simple but it does not provide a useful method for determining $P_G(x)$. The second is more complicated, but the steps of the proof provide a recursive method for calculating chromatic polynomials. We'll explore the method after the proof.

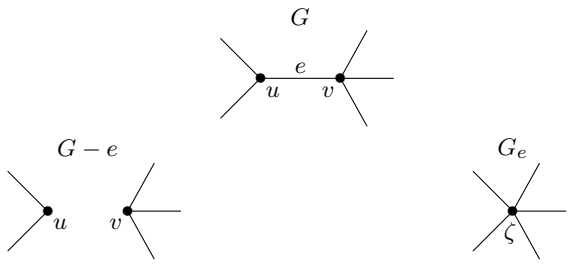


Figure 6.2 Forming $G - e$ and G_e from G by deletion and contraction.

Proof: (Nonconstructive) Let $d_G(k)$ be the number of ways to properly color the graph using k colors such that each color is used at least once. Clearly $d_G(k) = 0$ when $k > n$, the number of vertices of G. If we are given x colors, then the number of ways we can use exactly k of them to properly color G is $\binom{x}{k} d_G(k)$. (Choose k colors AND use them.) If $d_G(k) \neq 0$, this is a polynomial in x of degree k because $\binom{x}{k} = x(x-1)\cdots(x-k+1)/k!$, a polynomial in x of degree k. Since the number of colors actually used is between 1 and n,

$$P_G(x) \;=\; \sum_{k=1}^{n} \binom{x}{k} d_G(k), \tag{6.3}$$

a sum of a finite number of polynomials in x. Note that

- $d_G(n) \neq 0$, since it is always possible to color a graph with each vertex colored differently and
- the $k = n$ term in the sum (6.3) is the only term in the sum that has degree n.

Thus, $P_G(x)$ is a polynomial in x of degree n. (We needed to know that there was only one term of degree n because otherwise there might be cancellation, giving a polynomial of lower degree.) □

Proof: (Constructive) Let $G = (V, E)$. We'll use induction on $|E|$, the number of edges in the graph. If the graph has no edges, then $P_G(x) = x^n$ because any function from vertices to colors is acceptable as a coloring in this case.

You may find Figure 6.2 helpful. Suppose $e = \{u, v\} \in E$. Let $G - e = (V, E - e)$, a subgraph of G. This is called *deleting* the edge e. Every proper coloring of G is a proper coloring of $G - e$, but not conversely—a proper coloring λ of $G - e$ is a proper coloring of G if and only if $\lambda(u) \neq \lambda(v)$. A proper coloring λ of $G - e$ with $\lambda(u) = \lambda(v)$ can be thought of as a proper coloring of a graph G_e in which u and v have been identified. This is called *contracting* the edge e. Let's define G_e precisely. Choose $\zeta \notin V$, let $V_e = V \cup \{\zeta\} - \{u, v\}$ and let E_e be all two element subsets of V_e in E together with all sets $\{\zeta, y\}$ for which either $\{u, y\} \in E$ or $\{v, y\} \in E$ or both. The proper colorings of $G_e = (V_e, E_e)$ are in one-to-one correspondence with the proper colorings λ of $G - e$ for which $\lambda(u) = \lambda(v)$—we simply have $\lambda(\zeta) = \lambda(u) = \lambda(v)$. Thus every proper coloring of $G - e$ is a proper coloring of either G or G_e, but not both. By the Rule of Sum, $P_{G-e}(x) = P_G(x) + P_{G_e}(x)$ and so

$$P_G(x) \;=\; P_{G-e}(x) - P_{G_e}(x). \tag{6.4}$$

Since $G - e$ and G_e have less edges than G, it follows by induction that $P_{G-e}(x)$ is a polynomial of degree $|V| = n$ and that $P_{G_e}(x)$ is a polynomial of degree $|V_e| = n - 1$. Thus (6.4) is a polynomial of degree n. □

Example 6.8 Some chromatic polynomial calculations What is the chromatic polynomial of the graph with all $\binom{n}{2}$ possible edges present? In this case each vertex is connected to every other vertex by an edge so each vertex has a different color. Thus we get $x(x-1)(x-2)\cdots(x-n+1)$. The graph with all edges present is called the *complete graph* and is denoted by K_n.

What is the chromatic polynomial of the n vertex graph with no edges? We can color each vertex any way we choose, so the answer is x^n. By using the first proof of the theorem, we will obtain another formula for this chromatic polynomial. The graph can be colored using k colors (with each color used) by first partitioning the vertices into k blocks and then assigning a color to each block. By the Rule of Product, $d_G(k) = S(n,k)k!$, where $S(n,k)$ is a Stirling number of the second kind, introduced in Example 1.27. By the first proof and the fact that $P_G(x) = x^n$, we have

$$x^n \;=\; \sum_{k=1}^{n} \binom{x}{k} S(n,k)k! \;=\; \sum_{k=1}^{n} x(x-1)\cdots(x-k+1)\; S(n,k). \tag{6.5}$$

What is the chromatic polynomial of a path containing n vertices? Let the vertices be $\underline{n}$ and the edges be $\{i, i+1\}$ for $1 \le i < n$. Color vertex 1 using any of x colors. If the first i vertices have been colored, color vertex $i+1$ using any of the $x-1$ colors different from the color used on vertex i. Thus we see that the chromatic polynomial of the n vertex path is $x(x-1)^{n-1}$.

We now consider a more complicated problem. What is the chromatic polynomial of the graph that consists of just one cycle? Let n be the length of the cycle and let the answer be $C_n(x)$. Call the graph Z_n. Since Z_2 is just two connected vertices, $C_2(x) = x(x-1)$. It is easy to calculate $C_3(x)$: color one vertex arbitrarily, color the next vertex in a different color and color the last vertex different from the first two. Thus $C_3(x) = x(x-1)(x-2)$. What is $C_4(x)$? We use (6.4). If we delete an edge e from Z_4, we obtain a path on 4 vertices, which we have dealt with. If we contract e, we obtain Z_3, which we have dealt with. Thus

$$C_4(x) \;=\; x(x-1)^3 - x(x-1)(x-2).$$

What is the general formula? The previous argument generalizes to

$$C_n(x) \;=\; x(x-1)^{n-1} - C_{n-1}(x) \quad \text{for } n > 2. \tag{6.6}$$

How can we solve this recursion? This is not so clear. It is easier to see if we repeatedly use Figure 6.2 to expand things out until we obtain paths. The result for Z_5 is shown in the left side of Figure 6.3. In the right side of Figure 6.3, we have replaced each leaf by its chromatic polynomial. We can now work upwards from the leaves to compute the chromatic polynomial of the root.

This is a good way to do it for a graph that has no nice structure. In this case, however, we can write down the chromatic polynomial of the root directly. Notice that the kth leaf from the left (counting the leftmost as 0) has chromatic polynomial $x(x-1)^{n-k-1}$. If k is even it appears in the chromatic polynomial of the root with a plus sign, while if k is odd it appears with a minus sign. This shows that, for $n > 1$,

$$C_n(x) \;=\; \sum_{k=0}^{n-2} (-1)^k x(x-1)^{n-k-1} \;=\; x(x-1)^{n-1} \sum_{k=0}^{n-2} \left(\frac{1}{1-x}\right)^k,$$

which is a geometric series. Thus

$$C_n(x) \;=\; x(x-1)^{n-1} \frac{1 - \left(\frac{1}{1-x}\right)^{n-1}}{1 - \frac{1}{1-x}}.$$

You should show that this simplifies to

$$C_n(x) \;=\; (x-1)^n + (-1)^n(x-1) \tag{6.7}$$

for $n > 1$. □

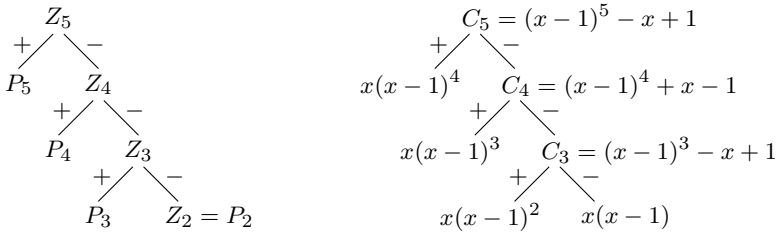


Figure 6.3 The calculation for $C_5(x)$ expanded. P_n is the n vertex path.

Exercises

6.2.1. Give an alternate proof of (6.7) by induction as follows: Prove by substitution that (6.7) satisfies (6.6) and show that (6.7) is correct for $n = 2$.

6.2.2. Conjecture and prove a formula for the chromatic polynomial of a tree. Your formula may include the number of vertices, the degrees of the vertices and anything else that you need. Be sure to indicate how you arrived at your conjecture. This formula can be useful in computing chromatic polynomials by the recursive method.
Hint. There is a simple formula.

6.2.3. The results in this exercise make it easier to calculate some chromatic polynomials using the recursive method.

(a) Suppose that G consists of two graphs H and K with no vertices in common. Prove that $P_G(x) = P_H(x)P_K(x)$.

(b) Suppose that G consists of two graphs H and K sharing exactly one vertex. Prove that $P_G(x) = P_H(x)P_K(x)/x$.

(c) Suppose that G is formed by taking two graphs H and K with no vertices in common, choosing vertices $v \in H$ and $w \in K$, and adding the edge $\{v, w\}$. Express $P_G(x)$ in terms of $P_H(x)$ and $P_K(x)$.

6.2.4. Let n and k be integers such that $1 < k < n-1$. Let G have $V = \underline{n}$ and edges $\{i, i+1\}$ for $1 \le i < n$, $\{1, n\}$, and $\{k, n\}$. Thus G is Z_n with one additional edge. Obtain a formula for $P_G(x)$.

6.2.5. Let L_n be the simple graph with $V = \underline{n} \times \underline{2}$ and $\{(i, j), (i', j')\}$ an edge if and only if $i = i'$ or $j = j'$ and $|i - i'| = 1$. The graph looks somewhat like a ladder and has $3n - 2$ edges. Prove that the chromatic polynomial of L_n is $(x^2 - 3x + 3)^{n-1}x(x - 1)$.

6.2.6. Let G be the simple graph with $V = \underline{3} \times \underline{3}$ and $\{(i, j), (i', j')\}$ an edge if and only if $|i-i'|+|j-j'| = 1$. It looks like a 2×2 board. Compute its chromatic polynomial.
Hint. It appears that any way it is done requires some computation. Using the edges joined to $(2, 2)$ for deletion and contraction is helpful.

*6.2.7. Construct a graph from the cube by removing the interior and the faces and leaving just the vertices and edges. What is the chromatic polynomial of this graph? (There is not some quick neat way to do this. Quite a bit of careful calculation is involved. A bit of care in selecting edges for removal and contraction will help.)

6.2.8. Give a proof of (6.5) by counting all functions from $\underline{n}$ to $\underline{x}$ in two ways.

6.2.9. Adapt the second proof that $P_G(x)$ is a polynomial of degree n to prove that the coefficients of the polynomial alternate in sign; that is, the coefficient of x^k is a nonnegative multiple of $(-1)^{n-k}$.

6.3 Planar Graphs

Recall that, drawing a graph in the plane without edges crossing is called embedding the graph in the plane. Any graph that can be embedded in the plane can be embedded in the sphere (i.e., the surface of a ball) and vice versa. The idea is simple: Cut a little hole out of the sphere in such a way that you don't remove any of the graph, then, pretending the sphere is a rubber sheet, stretch it flat to form a disc. Conversely, any map on the plane is bounded, so we can cut a disc containing a map out of the plane and curve it around to fit on a sphere. Thus, studying maps on the plane is equivalent to studying maps on the sphere.

Sometimes fairly simple concepts in mathematics lead to a considerable body of research. The research related to planar graphs is among the most accessible such bodies for someone without extensive mathematical training. Here are some of the research highlights and what we'll be doing about them.

1. The earliest is probably Euler's relation, which we'll discuss soon. If the sphere is cut along the edges of an embedded connected graph, we obtain pieces called *faces*. Euler discovered that the number of vertices and faces together differed from the number of edges by a constant. This has been extended to graphs embedded in other surfaces and to generalizations of graphs in higher dimensions. The result is an important number associated with a generalized surface called its *Euler characteristic*.
2. The four color problem has already been mentioned in the section on chromatic polynomials. As noted there, it has been generalized to other surfaces. We'll use Euler's relation to prove that five colors suffice on the plane.
3. A description of those graphs which can be drawn in the plane was obtained some time ago by Kuratowski: A graph is planar if and only if it does not "contain" either
   - K_5, the five vertex complete graph, or
   - $K_{3,3}$, the graph with $V = \{a_1, a_2, a_3, b_1, b_2, b_3\}$ and all nine edges of the form $\{a_i, b_j\}$.

   We say that G contains H if, except for labels we can obtain H from G by repeated use of the three operations:

   (a) delete an edge,

   (b) delete a vertex that lies on no edges and

   (c) if v lies only on the edges $e_1 = \{v, a_1\}$ and $e_2 = \{v, a_2\}$, delete v, e_1 and e_2 and add the edge $\{a_1, a_2\}$.

   Research has developed in two directions. One is algorithmic: Find good algorithms for deciding if a graph is planar and, if so, for embedding it in the plane. We'll discuss the algorithmic approach a bit. The other direction is more theoretical: Develop criteria like Kuratowski's for other surfaces. It has recently been proved that such criteria exist: There is always a finite list of graphs, like K_5 and $K_{3,3}$, that are bad. We will not be able to pursue this here; in fact, we will not even prove Kuratowski's Theorem.
4. Let's allow loops in our graphs. A graph embedded in the plane has a *dual*. A dual is constructed as follows for an embedded graph G. Place a vertex of $D(G)$ in each face of G. Thus there is a bijection between faces of G and vertices of $D(G)$. Every edge e of G is crossed by exactly one edge e' of $D(G)$, and vice versa, as follows. Let the face on one side of e be f_1 and the face on the other side be f_2. (Possibly, $f_1 = f_2$.) If v_1' and v_2' are the corresponding vertices in $D(G)$, then e' connects v_1' and v_2'. Attempting to extend the idea of a dual to other graphs leads to what are called "matroids" or "combinatorial geometries." We won't discuss this subject at all.

The algorithmic subsection does not require the earlier material, but it does require some knowledge of spanning trees, which were discussed in Section 6.1.

Our terminology "G contains H" in Item 3 is not standard. People are likely to say "G contains H homeomorphically." You should note that this is not the same as H being a subgraph of G. Repeated application of Rules (a) and (b) gives a subgraph of G. Conversely, all subgraphs of G can be obtained this way. Rule (c) allows us to contract an edge if it has an endpoint of degree 2. The result is not a subgraph of G. For example, applying (c) to a cycle of length 4 produces a cycle of length 3.

Euler's Relation

We'll state and prove Euler's relation and then examine some of its consequences.

Theorem 6.4 Euler's relation *Let $G = (V, E, \varphi)$ be a connected graph. Suppose that G has been embedded in the plane (or sphere) and that the embedding has f faces. Then*

$$|V| - |E| + f \;=\; 2. \tag{6.8}$$

This remains true if we extend the notion of a graph to allow loops.

Proof: In Exercise 5.4.3 (p. 140) you were asked to prove that $v = e + 1$ for trees. Since cutting along the edges of a tree does not make the plane (or sphere) fall apart, $f = 1$. Thus Euler's relation holds for all trees.

Is there some way we can prove the general result by using the fact that it is true for trees? This suggests induction, but on what should we induct? By Exercise 5.4.2 (p. 139), a tree has the least number of edges of any connected v-vertex graph. Thus we should somehow induct on the number of edges. With care, we could do this without reference to v, but there are better ways. One possibility is to induct on $d = e - v$, where $d \geq -1$. Trees correspond to the case $d = -1$ and so start the induction.

Another approach to the induction is to consider v to be fixed but arbitrary and induct on e. From this viewpoint, our induction starts at $e = v - 1$, which is the case of trees.

The two approaches are essentially the same. We'll take the latter approach.

Let $G = (V, E, \varphi)$ be any connected graph embedded in the plane. From now on, we will work in the plane, removing edges from this particular embedding of G. By Exercise 5.4.2, G contains a spanning tree $T = (V, E')$, say. Let $x \in E - E'$; that is, some edge of G not T. The subgraph G' induced by $E - \{x\}$ is still connected since it contains T.

Let $e' = e - 1$ and f' be the number of edges and faces of G'. By the induction assumption, G' satisfies (6.8) and so $v - e' + f' = 2$. We will soon prove that the opposite sides of x are in different faces of G. Thus, removing x merges these two faces and so $f' = f - 1$. This completes the inductive step.

We now prove our claim that opposite sides of x lie in different faces. Suppose $\varphi(x) = \{a, b\}$. Let P be a path from a to b in T. Adding x to the path produces a cycle C. Any face of the embedding must lie entirely on one side of C. Since one side of x is inside C and the other side of x is outside C, the two sides of x must lie in different faces. □

One interesting consequence of Euler's relation is that it tells us that no matter how we embed a graph $G = (V, E)$ in the plane—and there are often many ways to do so—it will always have $|E| - |V| + 2$ faces. We'll derive more interesting results.

Corollary 1 *If G is a planar connected simple graph with e edges and $v > 2$ vertices, then $e \leq 3v - 6$.*

Proof: When we trace around the boundary of a face of G, we encounter a sequence of vertices and edges, finally returning to our starting position. Call the sequence $v_1, e_1, v_2, e_2, \ldots, v_d, e_d, v_1$. We call d the *degree of the face*. Some edges may be encountered twice because both sides of them are on the same face. A tree is an extreme example of this: Every edge is encountered twice. Thus the degree of the face of a tree with e edges is $2e$. Let f_k be the number of faces of degree k. Since there are no loops or multiple edges, $f_1 = f_2 = 0$. If we trace around all faces, we encounter each edge exactly twice. Thus

$$2e \;=\; \sum_{k\geq 3} k f_k \;\geq\; \sum_{k\geq 3} 3 f_k = 3f$$

and so $f \leq 2e/3$. Consequently, $2 = v - e + f \leq v - e + 2e/3$. Rearrangement gives the corollary. □

Example 6.9 Nonplanarity of K_5 The graph K_5 has 5 vertices and 10 edges and so $3v - 6 = 9 < e$. Thus, it cannot be embedded in the plane. □

The following result will be useful in discussing coloring.

Corollary 2 *If G is a planar connected simple graph, then at least one vertex of G has degree less than 6.*

Proof: We suppose that the conclusion of the corollary is false and derive a contradiction. Let v_k be the number of vertices of degree k. Since each edge contains two vertices, $2e = \sum k v_k$. Since no vertex has degree less than 6, $v_k = 0$ for $k < 6$ and so

$$2e \;=\; \sum_{k\geq 6} k v_k \geq 6v.$$

Thus $e \geq 3v$, contradicting the previous corollary when $v \geq 3$. If $v \leq 3$, there are at most 3 edges, so the result is trivial. □

Exercises

6.3.1. Suppose that G is a planar connected graph with e edges and $v > 2$ vertices and that it contains no cycles of length 3. Prove that $e \geq 2v - 4$.

6.3.2. Prove that $K_{3,3}$ is not a planar graph. ($K_{3,3}$ was defined on page 162.)

6.3.3. We will call a connected graph embedded in the sphere a *regular graph* if all vertices have the same degree, say d_v and all faces have the same degree, say d_f.

(a) If e is the number of edges of the graph, prove that

$$e\left(\frac{2}{d_v} + \frac{2}{d_f} - 1\right) \;=\; 2. \qquad 6.9$$

(b) The possible graphs with $d_v = 2$ are simple to describe. Do it.

(c) By the previous part, we may as well assume that $d_v \geq 3$. Since both sides of (6.9) are positive, conclude that one of d_v and d_f must be 3 and that the other is a most 5.

(d) Draw all embedded regular graphs with $d_v > 2$.

6.3.4. Chemists have discovered a compound whose molecule is shaped like a hollow ball and consisting of **sixty** carbon atoms. It is called *buckminsterfullerene*. (This is *not* a joke.) There is speculation that this and similar carbon molecules may be common in outer space. A chemical compound can be thought of as a graph with the atoms as vertices and the chemical bonds as edges. Thus buckminsterfullerene can be viewed as a graph on the sphere. Because of the properties of carbon, it is reasonable to suppose that each atom is bound to exactly 3 others and that the faces of the associated embedded graph are either hexagons or pentagons. How many hexagons are there? How many pentagons are there?
Hint. One method is to determine the number of edges and then obtain two relations involving the number of pentagons and number of hexagons, one by counting edges and another from Euler's relation.

*6.3.5. A graph can be embedded on other finite surfaces besides the sphere. In this case, there is usually another condition: If we cut along all the edges of the graph, we get faces of the embedded graph and they all look like stretched polygons. This is called *properly* embedding the graph. To see that all embeddings are not proper, consider a torus (surface of a donut). A 3-cycle can be embedded around the torus like a bracelet. When we cut along the edges and straighten out the result, we have a cylinder, not a stretched polygon.

For proper embeddings in any surface, there is a relation like Euler's relation (6.8): $|V| - |E| + f = c$, but the value of c depends on the surface. For the sphere, it is 2.

(a) Properly embed some graph on the torus and compute c for it.

(b) Prove that your value in (a) is the same for all proper embeddings of graphs on the torus.
Hint. Cut around the torus like a bracelet, avoiding all vertices. Fill in each of the holes with a circle, introducing edges and vertices along the cuts.

The Five Color Theorem

Our goal is to prove the five color theorem:

Theorem 6.5 Heawood's Theorem *Every planar graph $G = (V, E)$ can be properly colored with five colors (i.e., adjacent vertices have distinct colors).*

Although four colors are enough, we will not prove that since the only known method is quite technical and requires considerable computing resources. On the other hand, if we were satisfied with six colors, the proof would much easier. We'll begin with it because it lays the foundation for five colors.

Proof: (Six colors) The proof will be by induction on the number of vertices of G.

A graph with at most six vertices can obviously be properly colored: Give each vertex a different color. Thus we can assume G has more than six vertices. We can also assume that G is connected because otherwise each component, which has less vertices than G, could be properly colored by the induction hypothesis. This would give a proper coloring of G.

Let $x \in V$ be a vertex of G with smallest degree. By Corollary 2, $d(x) \leq 5$. Let $G - x$ be the graph induced by $V - \{x\}$. By the induction hypothesis, $G - x$ can be properly 6-colored. Since there are at most 5 vertices adjacent to x in G, there must be a color available to use for x. □

This proof works for proving that G can be properly 5-colored except for one problem: The induction fails if x has degree 5 and the coloring of $G - x$ is such that the 5 vertices adjacent to x in G are all colored differently. We now show two different ways to get around this problem.

Proof: (Five colors, first proof) As noted above, we may assume that $d(x) = 5$. Label the vertices adjacent to x as $y_1, \ldots, y_5$, not necessarily in any particular order. Not all of the y_i's can be joined by edges because we would then have K_5 as a subgraph of G and, by Example 6.9, K_5 is not planar.

Suppose that y_1 and y_2 are not joined by an edge. Erase the edges $\{x, y_j\}$ from the picture in the plane for $j = 3, 4, 5$. Contract the edges $\{x, y_1\}$ and $\{x, y_2\}$. This merges x, y_1 and y_2 into a single vertex which we call y. We now have a graph H in the plane with two less vertices than G. By induction, we can properly 5-color H. Do so.

Color all vertices of G using the same coloring as for H, except for x, y_1 and y_2, which are colored as follows. Give y_1 and y_2 the same color as y. Give x a color different from all the colors used on the four vertices y, y_3, y_4 and y_5. (There must be one since we have five colors available.) Since H was properly colored and $\{y_1, y_2\}$ is not an edge of G, we have properly colored G. □

*Proof: (Five colors, second proof) As noted above, we may assume that $d(x) = 5$. Label the vertices adjacent to x as $y_1, \ldots, y_5$, reading clockwise around x. Properly color the subgraph induced by $V - x$. Let c_i be the color used for y_i. As noted above we may assume that the c_i's are distinct, for otherwise we could simply choose a color for x different from $c_1, \ldots, c_5$.

Call this position in the text HERE for future reference. Let H be the the subgraph of $G - x$ induced by the those vertices that have the same colors as y_1 and y_3. Either y_1 and y_3 belong to different components of H or to the same component. We consider the cases separately.

Suppose y_1 and y_3 belong to different components. Interchange c_1 and c_3 in the component containing y_1. We claim this is still a proper coloring of $G - x$. Why? First, it is still a proper coloring of the component of H containing y_1 and hence of H. Second, the vertices not in H are colored with colors other than c_1 and c_3, so those vertices adjacent to vertices in H remain properly colored. We have reduced this situation to the earlier case since only 4 colors are now used for the y_i.

Suppose that y_1 and y_3 belong to the same component. Then there is a path in H from y_1 to y_3. Add the edges $\{x, y_1\}$ and $\{x, y_3\}$ to the path to obtain a cycle. This cycle can be viewed as dividing the plane into two regions, the inside and the outside. Since y_2 and y_4 are on opposite sides of the cycle, any path joining them must contain a vertex on the cycle. Since all vertices on the cycle except y are colored c_1 or c_3, there is no path of vertices colored c_2 and c_4 in $G - x$ joining y_2 and y_4. Now go back to HERE, using the subscripts 2 and 4 in place of 1 and 3. Since y_2 and y_4 will belong to different components, we will not return to this paragraph. Thus, the proof is complete. □

Exercises

6.3.6. Let G be the simple graph with $V = \underline{7}$ and edge set

$$E = \{\{1,2\},\{1,3\},\{1,4\},\{1,7\},\{2,3\},\{2,4\},$$
$$\{2,5\},\{2,6\},\{2,7\},\{3,4\},\{4,7\},\{5,6\},\{6,7\}\}.$$

(a) Embed G in the plane.

(b) The vertices of V have a natural ordering. Thus a function specifying a coloring of V can be written in one-line form. In this one-line form, what is the lexicographically least proper coloring of G when the available "colors" are a, b, c, d, e and f?

(c) What is the lexicographically least proper coloring of G using the colors a, b, c and d? Notice that the lexicographically least proper coloring in (b) uses five colors but there is another coloring that uses only four colors in (c).

6.3.7. Prove that if G is a planar graph with $V = \underline{5}$, then the lexicographically least proper coloring of G using the colors a, b, c, d and e uses only 4 colors.

6.3.8. Suppose $V = \underline{6}$ and the available colors are a, b, c, d, e. Find a planar graph G with vertex set V such that the lexicographically least proper coloring of G is not a four coloring of G.

*6.3.9. What's wrong with the following idea for showing that 4 colors are enough? By the argument for 5 colors, our only problem is a vertex of degree 5 or less such that the adjacent vertices require 4 colors. Use the idea in second argument in the text. With a vertex of degree 4, the argument can be used as stated in the text with all mention of y_5 deleted.

Now suppose the vertex has degree 5. The argument in the text guarantees that $y_1, \ldots, y_5$ can be colored with 4 colors. Do so. We can still select two vertices that are colored differently and are not adjacent in the clockwise listing. Thus the argument given in the text applies and we can reduce the number of colors needed for $y_1, \ldots, y_5$ from 4 to 3.

*6.3.10. In Exercise 6.3.5 we looked at proper embeddings of graphs in a torus. It can be shown that every embedding (proper or not) of a graph in a torus can be properly colored using at most seven colors. Find a graph embedded in a torus that requires seven colors.
Hint. There is one with just seven vertices.

*Algorithmic Questions

We do not know of an algorithm for 4 coloring a planar graph in a reasonable time. The first proof of the Five Color Theorem leads to a reasonable algorithm for coloring a planar graph with 5 colors. At each step, the number of vertices is decreased by 2. Once the reduced graph is colored, it is a simple matter to color the given graph: Adjust the previous coloring in[1] $O^+(1)$ time by assigning colors to the three vertices x, y_1 and y_2 that were merged to form y. The time required is $O^+(1)$. If $R(n)$ is the maximum time needed to reduce an n vertex planar graph, then The total time is[2]

$$R(n) + R(n-2) + \cdots + O^+(n),$$

where the $O^+(n)$ is due to the roughly $n/2$ times the coloring must be adjusted. Note that $R(n)$ comes primarily from finding a vertex of degree 5 or less. It should be fairly clear to you that $R(n)$ is $O^+(n)$ and so the time for coloring is $O^+(n^2)$. One would expect that using a sophisticated data structure to represent the graph would allow us to improve this time. We will not pursue this here.

We now turn our attention to the problem of deciding if a graph is planar. At first glance, there is no obvious algorithm for doing this except to use Kuratowski's Theorem. This involves a tremendous amount of computer time because there are so many possible choices for the operations (a)–(c) described at the start of Section 6.3. Because of this, it is somewhat surprising that there are algorithms with worst case running times that are $\Theta(|V|)$. We'll examine some of the ideas associated with one such algorithm here but will not develop the complex data structures needed to achieve $\Theta(|V|)$ running time.

To check if a graph is planar, it suffices to merge multiple edges and to check that each connected component is planar. This should be obvious. A bit less obvious is the fact that it suffices to check each biconnected component. (This was defined in Example 6.4 (p. 154).) To see this, note that we can begin by embedding one bicomponent in the plane. Suppose that some subgraph H of the given graph has been already embedded. Any unembedded bicomponent C that shares a vertex v with H can then be embedded in a face of H adjacent to v in such a way that the v of C and the v of H can be merged into one vertex. This process can be iterated. Because of this, we limit our attention to simple biconnected graphs.

[1] The notation for $O^+(\,)$ is discussed in Appendix B.
[2] Recall that $f(n) + O^+(n)$ means $f(n)$ plus some function that is in $O^+(n)$.

Definition 6.4 ***st*-labeling** *Let $G = (\underline{n}, E)$ be a simple biconnected graph with $\{s,t\} \in E$, an* ***st*-labeling** *of G is a permutation λ of $\underline{n}$ such that*

(a) $\lambda(s) = 1$,

(b) $\lambda(t) = n$ *and,*

(c) *whenever $1 < \lambda(v) < n$, there are vertices u and w adjacent to v such that $\lambda(u) < \lambda(v) < \lambda(w)$.*

We will soon prove that such a vertex labeling exists and give a method for finding it, but first, we explain how to use it.

Start embedding G in the plane by drawing the edge $\{s,t\}$. Suppose that we have managed to embed the subgraph H_k induced by the vertices with $\lambda(v) < k$ together with t. If $\lambda(x) = k$, then we must embed x in the same face as t. Why is this? By (c), there exist vertices $x = w_1, w_2, \ldots = t$ such that $\{w_i, w_{i+1}\}$ is an edge and $\lambda(w_i)$ is an increasing function of i. Since none of these vertices except t have been embedded, they must all lie in the same face of H_k. Thus we know which face of H_k to put x in. Unfortunately, parts of our embedding of H_k may be wrong in the sense that we cannot now embed x to construct H_{k+1}. How can we correct that?

The previous observation implies that we can start out by placing the vertices of G in the plane so that v is at $(\lambda(v), 0)$. Correcting the problems with H_k can be done by redrawing edges without moving the vertices. There are systematic, but rather complicated, ways of finding a good redrawing of H_k (or proving that none exists if the graph is not planar). We will not discuss them. For relatively small graphs, this can be done by inspection.

We now present an algorithm for finding an *st*-labeling. The validity of the algorithm implies that such a labeling exists, thus completing the proof of our planarity algorithm. We'll find an injection $\lambda: \underline{n} \to \mathbb{R}$ that satisfies (a)–(c) in Definition 6.4. It is easy to construct an *st*-labeling from such a λ by replacing the kth smallest value of λ with k. Suppose that we specified λ for a subgraph G_Z of G induced by some subset Z of the vertices of G. We assume that $s, t \in Z$ and that λ satisfies (a)–(c) on G_Z. Suppose that $\{x,y\}$ is not in G_Z. Since G is biconnected, there is a cycle in G containing $\{x,y\}$ and $\{s,t\}$. This means that there are disjoint paths in G either

(i) from s to x and from t to y or

(ii) from s to y and from t to x.

If (ii) holds, interchange the meanings of x and y to convert it to (i). Thus we may assume that (i) holds.

Let the paths in (i) be $s = u_1, u_2, \ldots = x$ and $t = w_1, w_2, \ldots = y$. Suppose that u_i and w_j are the last vertices in Z on these two paths. Let v_k be the kth vertex on the path

$$u_i, u_{i+1}, \ldots, x, y, \ldots, w_{j+1}, w_j \tag{6.10}$$

and let m be the total number of vertices on the path. Except for u_i and w_j, none of the vertices in (6.10) are in Z. Add them to Z and define $\lambda(v_k)$ for $k \neq 1, m$ such that λ is still an injection and $\lambda(v_k)$ is monotonic on the path. In other words, if $\lambda(u_i) < \lambda(w_j)$, then λ will be strictly increasing on the path, and otherwise it will be strictly decreasing. Making λ an injection is easy since there are an infinite number of real numbers between $\lambda(u_i)$ and $\lambda(w_j)$. We leave it to you to show that this function satisfies (c) for the vertices that were just added to Z.

Example 6.10 Let G be the simple graph with $V = \underline{7}$ and edge set

$$E = \{\{1,2\},\{1,3\},\{1,4\},\{1,7\},\{2,3\},\{2,4\},\\ \{2,5\},\{2,6\},\{2,7\},\{3,4\},\{4,7\},\{5,6\},\{6,7\}\}.$$

We'll use the algorithm to construct a 2,5-labeling for it. The labeling λ will be written in two line form with blanks for unassigned values; i.e., for vertices not yet added to Z. To begin with $Z = \{2,5\}$ and

$$\lambda = \left(\begin{smallmatrix} 1&2&3&4&5&6&7\\ 1& & & &7& & \end{smallmatrix}\right).$$

Let $\{x,y\} = \{2,6\}$. In this case, a cycle is 2,6,5. Thus $s = u_1 = x$, $t = w_1$ and $w_2 = y$. Hence $i = 1$, $j = 1$ and the path (6.10) is $s = x, y, t = 2, 6, 5$. We must choose $\lambda(6)$ between 1 and 7, say 4. Thus

$$\lambda = \left(\begin{smallmatrix} 1&2&3&4&5&6&7\\ 1& & & &7&4& \end{smallmatrix}\right).$$

Let $\{x,y\} = \{3,4\}$. A cycle is $3,4,7,6,5,2$ and the path (6.10) is $2,3,4,7,6$. We must choose $\lambda(x)$, $\lambda(y)$ and $\lambda(7)$ between 1 and 4 and monotonic increasing. We take

$$\lambda = \left(\begin{smallmatrix} 1&2&3&4&5&6&7\\ 1&2&3&7&4&3.5& \end{smallmatrix}\right).$$

Finally, with $\{x,y\}$, a cycle is $2,1,4,7,6,5$ and the path (6.10) is 2,1,4. We take

$$\lambda = \left(\begin{smallmatrix} 1&2&3&4&5&6&7\\ 2.5&1&2&3&7&4&3.5 \end{smallmatrix}\right).$$

Adjusting to preserve the order and get a permutation, we finally have

$$\lambda = \left(\begin{smallmatrix} 1&2&3&4&5&6&7\\ 3&1&2&4&7&6&5 \end{smallmatrix}\right). \quad \square$$

This algorithm for deciding planarity can be adapted to produce an actual embedding in the plane, if the graph is planar. In practical problems, one usually imposes other constraints on the embedding. For example, if one is looking at pictures of graphs, the embedding should spread the vertices and edges out in a reasonably nice fashion. On the other hand, in VLSI design, one often assumes that the maximum degree of the vertices is at most four and requires that the edges be laid out on a regular grid. Other VLSI layout problems involve vertices which take up space on the plane and various edge constraints. In VLSI design, one would also like to keep the edge lengths as small as possible. A further complication that arises in VLSI is that the graph may not be planar, but we would like to draw it in the plane with relatively few crossings. These are hard problems—often they are "NP-hard," a term discussed in Section B.3 (p. 377).

Exercises

6.3.11. Let G be the simple graph it $V = \underline{7}$ and edge set

$$E = \{\{1,2\},\{1,3\},\{1,4\},\{1,7\},\{2,3\},\{2,4\},\\ \{2,5\},\{2,6\},\{2,7\},\{3,4\},\{4,7\},\{5,6\},\{6,7\}\}.$$

(a) Use the algorithm in the text to construct a $1,7$-labeling for G.

(b) Use the algorithm in the text to construct a $7,1$-labeling for G.

6.3.12. Construct an st-labeling for K_5. (Since K_5 is completely symmetric, it doesn't matter what vertices you choose for s and t.)

6.3.13. Construct an st-labeling for $K_{3,3}$. (Again, the symmetry guarantees that it doesn't matter which edge is chosen for $\{s,t\}$.)

1	2, 7, 12	11	9, 12, 13, 14	21	20, 22, 23, 24
2	1, 6, 3, 10	12	11, 9, 1	22	21, 20, 3
3	2, 4, 17, 22, 25	13	11, 14, 15	23	21, 24
4	3, 5, 29, 24, 27	14	13, 11	24	23, 21, 4
5	4, 6, 28, 30	15	13, 7	25	6, 26, 27
6	5, 20, 25, 7, 2	16	7, 18, 19, 17	26	3, 25
7	6, 8, 16, 1, 19, 15	17	16, 18, 3	27	4, 25
8	7, 9, 10	18	17, 16	28	5, 29, 30
9	10, 12, 11, 8	19	16, 7	29	4, 28
10	8, 9, 1	20	6, 21, 22	30	5, 28

Figure 6.4 The adjacency lists of a simple graph for Exercise 6.3.16.

6.3.14. We return to Exercise 5.5.11: Suppose that G is a connected graph with no isthmuses. We want to prove that the edges of G can be directed so that the resulting directed graph is strongly connected.

(a) Suppose that G is biconnected and has at least two edges. Use st-labelings to prove that the result is true in this case.

(b) Prove if a graph has no isthmuses, then every bicomponent has at least two edges.

(c) Complete the proof of this exercise.

6.3.15. Prove that a graph is biconnected if and only if it has an st-labeling.
Hint. Given any edge different from $\{s, t\}$, use the properties of an st-labeling to find a cycle containing that edge and $\{s, t\}$.

*6.3.16. $G = (\underline{30}, E)$ is a simple graph given by Figure 6.4. Row k lists the vertices of G that are adjacent to k. Embed G in the plane using the algorithm described in the text. Produce a 5 coloring of G and, if you can, make it a 4 coloring.

6.4 Flows in Networks

We now discuss "flows in networks," an application of directed graphs. Examples of this concept are fluid flowing in pipes, traffic moving on freeways and telephone conversations through telephone company circuits. We'll use fluid in pipes to motivate and interpret the mathematical concepts.

The Concepts

In Figure 6.5 we see a simple directed graph. Note that its edges are labeled; e.g., the directed edge (D_1, P_1) has label 2. Imagine that the directed edges of the graph represent pipes through which a fluid is flowing. A label on an edge represents the rate (measured in liters per second) at which the fluid is flowing along the pipe represented by that edge. We denote this flow rate function by f. Thus, $f(D_1, P_1) = 2$ liters/sec is an example of a value of f in Figure 6.5.

The vertices V in Figure 6.5 are divided into two classes,

$$\mathcal{D} = \{D_1, D_2, D_3, D_4\} \quad \text{and} \quad \mathcal{P} = \{P_1, P_2, P_3, P_4\}.$$

Think of the vertices in $\mathcal{D}$ as depots and those in $\mathcal{P}$ as pumps. Fluid can enter or leave the system at a depot but not at a pump. This corresponds to practical experience with pumps: If the rate at which fluid is flowing into a pump exceeds that at which it is flowing out, the pump will rupture, while, if the inflow is less than the outflow, the pump must be creating fluid.

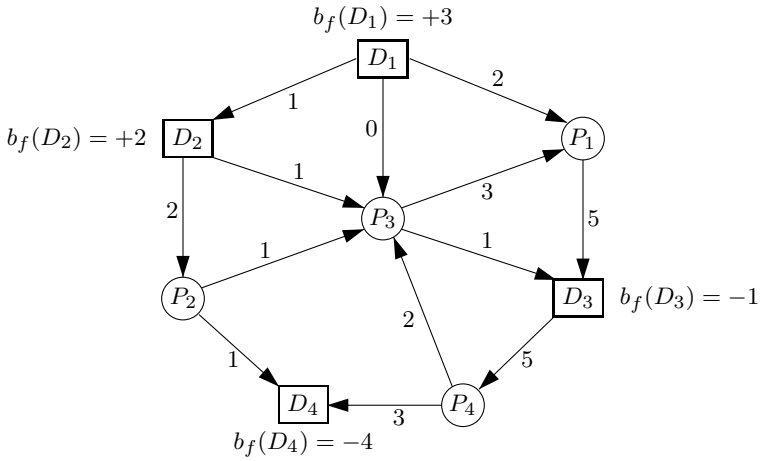


Figure 6.5 A network of pipes with depots D_i, pumps P_i and flow function f (on edges).

We now associate with each vertex $v \in V$ a number $b_f(v)$ which measures the balance of fluid flow at the vertex: $b_f(v)$ equals the sum of all the flow rates for pipes out of v minus the sum of all the flow rates for pipes into v. By the previous paragraph, $b_f(v) = 0$ if $v \in \mathcal{P}$. In Figure 6.5, the nonzero values of b_f are written by the depots. The fact that $b_f(D_2) = +2$ means that we must constantly pump fluid into D_2 from outside the system at a rate of 2 liters/sec to keep the flow rates as indicated. Likewise, we must constantly extract fluid from D_4 at the rate of 4 liters/sec to maintain stability.

It is useful to summarize some of the above ideas with a precise definition. Note that we do not limit ourselves to directed graphs which are simple.

Definition 6.5 Flow in a digraph *Let $G = (V, E, \varphi)$ be a directed graph. For $v \in V$, define*

$$\mathrm{IN}(v) \;=\; \{e \in E : \varphi(e) = (x, v) \text{ for some } x \in V\}$$

and

$$\mathrm{OUT}(v) \;=\; \{e \in E : \varphi(e) = (v, y) \text{ for some } y \in V\}.$$

Let f be a function from E to the nonnegative real numbers; i.e., $f\colon E \to \mathbb{R}^+$. Define $b_f\colon V \to \mathbb{R}$ by

$$b_f(v) \;=\; \sum_{e \in \mathrm{OUT}(v)} f(e) \;-\; \sum_{e \in \mathrm{IN}(v)} f(e).$$

Let $(\mathcal{D}, \mathcal{P})$ be an ordered partition of V into two sets. The function f will be called a **flow** *with respect to this partition if $b_f(v) = 0$ for all $v \in \mathcal{P}$. We call the function b_f the* **balance** *of the flow f.*

You may have noticed that our discussion of flows in networks is missing something important, namely the capacities of the pipes to carry fluid. In Figure 6.6, we have included this information. Attached to each edge is a dotted semicircle containing the maximum amount of fluid in liters/sec that can flow through that pipe. This is the *capacity* of the edge (pipe) and is denoted by c; e.g., $c(P_1, D_3) = 6$ in Figure 6.6. The capacity c is a function from E to the set of positive real numbers. We are interested in flows which do not exceed the capacities of the edges. Realistically, it would also be necessary to specify capacities for the pumps and depots. We will do that in the exercises, but for now we'll assume they have a much larger capacity than the pipes and so can handle any flow that the pipes can.

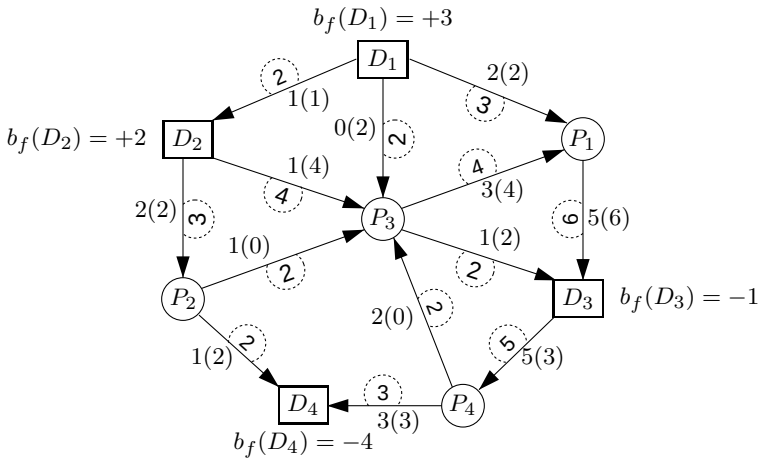


Figure 6.6 The network in Figure 6.5 with capacities c (in dotted semicircles), the flow from Figure 6.5, and another admissible flow p (in parentheses).

The set $\mathcal{D}$ will now be divided into two subsets called the *sources*, where fluid can enter the network, and the *sinks*, where fluid can leave the network. Our goal is to maximize the rate at which fluid passes through the network; that is to maximize the sum of $b_f(v)$ over all sources. We'll present a formal definition and then look at how it applies to Figure 6.6. Since we will spend some time working with Figure 6.6, you may find it useful to make some copies of it for scratch work.

Definition 6.6 Some network flow terminology *Let $G = (V, E, \varphi)$ be a directed graph. Let c be a function from E to the nonnegative reals, called the* **capacity** *function for G. Let f be a flow function on the directed graph $G = (V, E, \varphi)$ with vertex partition $(\mathcal{D}, \mathcal{P})$ and balance function b_f, as defined in Definition 6.5. The flow f will be called* **admissible** *with respect to the capacity function c if $f(e) \leq c(e)$ for all edges e. Let $(\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}})$ be an arbitrary ordered partition of $\mathcal{D}$ into two nonempty sets. We call $\mathcal{D}_{\text{in}}$ the set of* **source vertices** *for this partition and $\mathcal{D}_{\text{out}}$ the set of* **sink vertices**. *We define the* **value** *of f with respect to this partition to be*

$$\text{value}(f) \;=\; \sum_{v \in \mathcal{D}_{\text{in}}} b_f(v).$$

An admissible flow f will be called **maximum** *with respect to $(\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}})$ if $\text{value}(g) \leq \text{value}(f)$ for all other admissible flows g.*

In general, the partitioning you are given of the set $\mathcal{D}$ of depots into the two subsets $\mathcal{D}_{\text{in}}$ and $\mathcal{D}_{\text{out}}$ is completely arbitrary. Once this is done, the two sets will be kept the same throughout the problem of maximizing value(f). It is sometimes convenient to write $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$ to refer to the graph G with the capacity function c and the "source-sink" partition $(\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}})$. Our basic problem is, given $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$, find an admissible flow that is maximum.

In Figure 6.6 let $\{D_1, D_2\} = \mathcal{D}_{\text{in}}$ and $\{D_3, D_4\} = \mathcal{D}_{\text{out}}$. Intuitively, value($f$) is the amount of fluid in liters/sec that must be added to D_1 and D_2 to maintain the flow f. (The same amount overflows at D_3 and D_4.) You have to be careful though! If you just pick some admissible flow f without worrying about maximizing value(f), you might pick one with value(f) < 0, in which case fluid will have to be extracted from the source $\mathcal{D}_{in}$ to maintain the flow. It is only for the flow f that maximizes value(f) that we can be sure that fluid is added to the source vertices (or at least not extracted).

An Algorithm for Constructing a Maximum Flow

Now we'll study Figure 6.6 more carefully to help us formulate an algorithm for finding an admissible flow function f which maximizes $\text{value}(f) = b_f(D_1) + b_f(D_2)$, where b_f is the balance function of the function f.

In addition to the capacities shown on the edges of Figure 6.6 there are two sets of numbers. One number has parentheses around it, such as (4) on the edge (D_2, P_3). The other number does not have parentheses, such as 1 on (D_2, P_3). The parenthesized numbers define a flow, which we call p for "parentheses," and the other numbers define a flow which we call f. Referring to the edge $e = (D_2, P_3)$, $f(e) = 1$ and $p(e) = 4$. You should check that f and p satisfy the definitions of a flow with respect to the depots $\mathcal{D}$ and pumps $\mathcal{P}$. Computing the values of f and p with respect to $\mathcal{D}_{\text{in}} = \{D_1, D_2\}$ we obtain $\text{value}(f) = 3 + 2 = 5$ and $\text{value}(p) = 5 + 5 = 10$. Thus p has the higher value. In fact, we will later prove that p is a flow of maximum value with respect to the set of sources $\mathcal{D}_{\text{in}}$.

To begin with, concentrate on the flow f. Go to Figure 6.6 and follow the edges connecting the sequence of vertices (D_2, P_3, P_4, D_3). They form an (undirected) path on which you first go forward along the edge (D_2, P_3), then backwards along the edge (P_4, P_3), then backwards along the edge (D_3, P_4). Note that for each forward edge e, $f(e) < c(e)$ and for each backward edge e, $f(e) > 0$. These conditions, $f(e) < c(e)$ on forward edges and $f(e) > 0$ on backward edges are very important for the general case, which we discuss later.

For each forward edge e, define $\delta(e) = c(e) - f(e)$. For each backward edge e, define $\delta(e) = f(e)$. In our particular path we have $\delta(D_2, P_3) = \delta(P_4, P_3) = \delta(D_3, P_4) = 2$. Let δ denote the minimum value of $\delta(e)$ over all edges in the path. In our case $\delta = 2$. We now define a new flow g based on f, the path, and δ:

- For each forward edge e on the path, add δ to $f(e)$ to get $g(e)$.
- For each backward edge e on the path, subtract δ from $f(e)$ to get $g(e)$.
- For all edges e not on the path, $g(e) = f(e)$.

This process is called "augmenting the flow" along the path. You should convince yourself that in our example, $\text{value}(g) = \text{value}(f) + \delta = \text{value}(f) + 2$. This type of relation will be true in general.

If you now study Figure 6.6, you should be able to convince yourself that there is no path from $\mathcal{D}_{\text{in}}$ to $\mathcal{D}_{\text{out}}$ along which p can be augmented. This observation is the key to proving that p is a maximum flow: We will see that maximum flows are those which cannot be augmented in this way.

We are now ready for some definitions and proofs. The next definition is suggested by our previous discussion of augmenting flows.

Definition 6.7 *Let f be an admissible flow function for $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$. Suppose that $(v_1, v_2, ..., v_k)$ is an undirected path in G with $v_1 \in \mathcal{D}_{\text{in}}$ and $v_i \notin \mathcal{D}_{\text{in}}$ for $i > 1$. If for each forward edge e in this path, $f(e) < c(e)$ and for each backward edge e, $f(e) > 0$ then we say that the path is* **augmentable**. *If in addition $v_k \in \mathcal{D}_{\text{out}}$ then we say that the path is a* **complete augmentable path**. *Let $\delta(e) = f(e)$ if e is a backward edge on the path and let $\delta(e) = c(e) - f(e)$ if e is a forward edge. The minimum value of $\delta(e)$ over all edges of the path, denoted by δ, will be called the* **increment of the path**. *Let $\mathcal{A}(f)$ be those vertices that lie on some augmentable path of f, together with all the vertices in $\mathcal{D}_{\text{in}}$.*

In Figure 6.6, with respect to the admissible flow f, the path (D_1, P_1, P_3) is augmentable. The path $(D_2, P_2, P_3, P_4, D_3)$ is a complete augmentable path. So is the path (D_2, P_3, P_4, D_3).

Theorem 6.6 Augmentable Path Theorem *A flow f is a maximum flow if and only if it has no complete augmentable path; that is, if and only if $\mathcal{A}(f) \cap \mathcal{D}_{\text{out}} = \emptyset$.*

Proof: If such a complete augmentable path exists, use it to augment f by δ and obtain a new flow p. Since the first vertex on the path lies in $\mathcal{D}_{\text{in}}$, it follows that $\text{value}(p) = \text{value}(f) + \delta > \text{value}(f)$. Therefore f was not a maximum flow.

Now suppose that no complete augmentable path exists for the flow f. Let $A = \mathcal{A}(f)$ and $B = V - A$. We will now consider what happens to flows on edges between A and B. For this purpose, it will be useful to have a bit of notation. For C and D subsets of V, let $\text{FROM}(C, D)$ be all $e \in E$ with $\varphi(e) \in C \times D$; i.e., the edges from C to D. We claim:

1. For *any* flow g,
$$\text{value}(g) = \sum_{e \in \text{FROM}(A,B)} g(e) - \sum_{e \in \text{FROM}(B,A)} g(e). \tag{6.11}$$
2. For the flow f, if $e \in \text{FROM}(A, B)$ then $f(e) = c(e)$, and if $e \in \text{FROM}(B, A)$ then $f(e) = 0$.

The proofs of the claims are left as exercises. Suppose the claims are proved. Since $0 \leq g(e) \leq c(e)$ for any flow g, it follows that

$$\begin{aligned} \text{value}(g) &= \sum_{e \in \text{FROM}(A,B)} g(e) - \sum_{e \in \text{FROM}(B,A)} g(e) \\ &\leq \sum_{e \in \text{FROM}(A,B)} c(e) - \sum_{e \in \text{FROM}(B,A)} 0 \\ &= \text{value}(f). \end{aligned}$$

Since g was *any* flow whatsoever, it follows that f is a maximum flow. □

This theorem contains the general idea for an algorithm that computes a maximum flow for $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$. The first thing to do is to choose an admissible flow f. The flow $f(e) = 0$ for all e will always do. Usually, by inspection, you can do better than that. The general idea is that, given a flow f such that $\mathcal{A}(f) \cap \mathcal{D}_{\text{out}} \neq \emptyset$, we can find a complete augmentable path and use that path to produce a flow of higher value. Here's the procedure

```
/* The main procedure */
Procedure maxflow
     Set f(e) = 0 for all e.
     While A(f) ∩ D_out ≠ ∅, augment(f).
     Return f.
End

/* Replace f with a bigger flow. */
Procedure augment(f)
     Find a complete augmentable path (v_1, v_2, ..., v_k).
     Compute the increment δ of this path.
     If e is a forward edge of the path, set f(e) = f(e) + δ.
     If e is a backward edge of the path, set f(e) = f(e) − δ.
     Return f.
End
```

We have left it up to you to do such nontrivial things as decide if $\mathcal{A}(f) \cap \mathcal{D}_{\text{out}}$ is empty and to find a complete augmentable path. In the examples and exercises in this book it will be fairly easy to do these things. On larger scale problems, the efficiency of the algorithms used for these things can be critical. This is a topic for a course in data structures and/or algorithm design and is beyond the scope of this book.

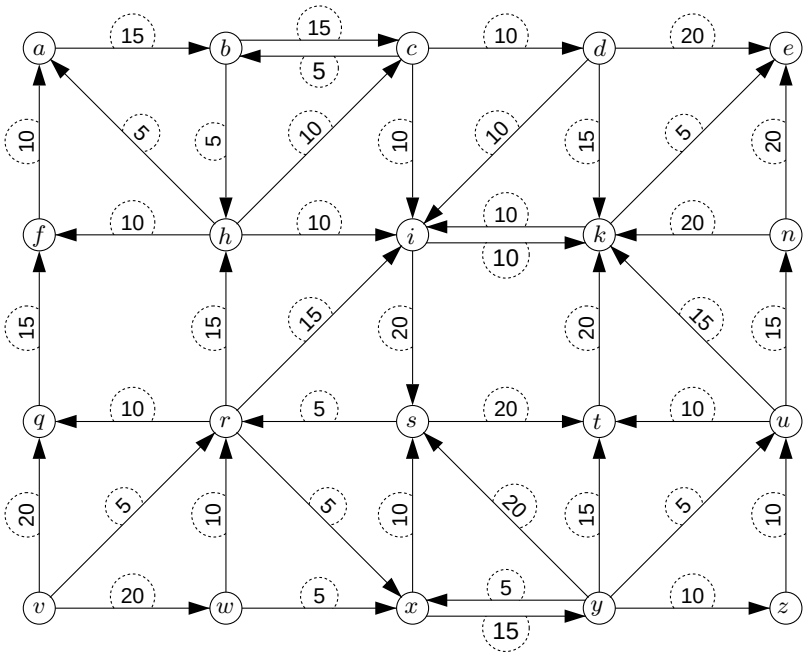


Figure 6.7 The network for Exercise 6.4.1.

It is possible that, like Zeno's paradox, our algorithm will run forever: `augment` could simply produce a flow f with value(f) halfway between the value of the flow it was given and the value of a maximum flow. In fact, the algorithm will always stop. We'll prove a weaker version:

Theorem 6.7 Integer Flow Theorem *If the capacities of a network are all integers, then the* `maxflow` *algorithm stops after a finite number of steps and produces a maximum flow which assigns integer flows to the edges.*

Proof: We claim that the calculations in the algorithm involve only integer values for f and δ. This can be proved by induction: Before any iterations, f is an integer valued function. Suppose that we call `augment`(f) with an integer valued f. Since δ is a minimum of numbers of the form $f(e)$ and $c(e) - f(e)$, which are all integers, δ is a positive integer. Thus the new f is integer valued and has a value at least one larger than the old f. Thus, after n steps, value(f) $\geq n$. If a maximum flow has value F, then a maximum flow is reached after at most F steps. □

Although this algorithm stops, it is a poor algorithm. Quite a bit of work has been done on finding fast network flow algorithms. Unfortunately, improvements usually lead to more complex algorithms that use more complicated data structures. One easy improvement is to use a shortest complete augmentable path in `augment`. This leads to an easily programmed algorithm which often runs fairly quickly. Another poor feature of our algorithm lies in the fact that all the calculations needed to find an augmentable path are thrown away after the path has been found. With just a little more work, one may be able to do several augmentations at the same time. The worst case behavior of the resulting algorithm is good, namely $O(|V|^3)$. Unfortunately, it is rather complicated so we will not discuss it here.

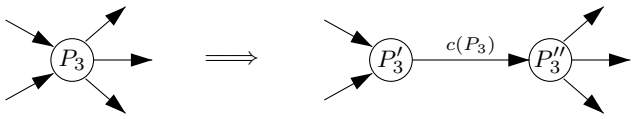


Figure 6.8 Converting pump capacity to edge capacity for Exercise 6.4.4.

Exercises

6.4.1. The parts of this problem all refer to Figure 6.7.

(a) With $\mathcal{D}_{\text{in}} = \{v, w, x, y, z\}$ and $\mathcal{D}_{\text{out}} = \{a, b, c, d, e\}$, find a maximum flow. Also, find a minimum cut set and verify that its capacity is equal to the value of your maximum flow.

(b) Returning to the previous part, find a different maximum flow and a different minimum cut set.

(c) With $\mathcal{D}_{\text{in}} = \{v\}$ and $\mathcal{D}_{\text{out}} = \{e\}$, find a maximum flow. Also, find a minimum cut set and verify that its capacity is equal to the value of your maximum flow.

(d) In the previous part, is the maximum flow unique? Is the minimum cut set unique?

6.4.2. Prove Claim 2 in the proof of the Augmentable Path Theorem.
Hint. Prove that $f(e) < c(e)$ for $e \in \text{FROM}(A, B)$ implies that the ends of e are in A, a contradiction. Do something similar for $\text{FROM}(B, A)$.

6.4.3. Prove (6.11).
Hint. Prove that $\text{value}(g) = \sum_{v \in A} b(v)$ and that each edge e with both ends in A contributes both $c(e)$ and $-c(e)$ to $\sum_{v \in A} b(v)$.

6.4.4. We didn't consider the capacities of the pumps in dealing with Figure 6.6. Essentially, we assumed that the pumps could handle any flow rates that might arise. Suppose that pumps P_1, P_2 and P_3 are having mechanical problems and can only pump 3 liters/sec. What is a maximum flow for $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$ with $\mathcal{D}_{\text{in}} = \{D_1, D_2\}$ and $\mathcal{D}_{\text{out}} = \{D_3, D_4\}$? Figure 6.8 shows how to convert a pump capacity into an edge capacity.

6.4.5. Consider again the network flow problem of Figure 6.6. The problem was defined by $N = (G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$ where $\mathcal{D}_{\text{in}} = \{D_1, D_2\}$ and $\mathcal{D}_{\text{out}} = \{D_3, D_4\}$. Imagine that two new depots are created as shown in Figure 6.9, and all of the original depots are converted into pumping stations. Let $N' = (G', c', (\mathcal{D}'_{\text{in}}, \mathcal{D}'_{\text{out}}))$ denote this new problem, where $\mathcal{D}'_{\text{in}} = \{D_0\}$ and $\mathcal{D}'_{\text{out}} = \{D_5\}$.

(a) What are the smallest values of $c'(D_0, P'_1) = c'_1$, $c'(D_0, P'_2) = c'_2$, $c'(P'_3, D_5) = c'_3$ and $c'(P'_4, D_5) = c'_4$ that guarantees that if p' is a maximum flow for N' then p' restricted to the edges of G is a maximum flow for N? Explain.

(b) With your choices of c'_i, will it be true that any maximum flow on N can be used to get a maximum flow on N'? Explain.

*Cut Partitions and Cut Sets

The "Max-Flow Min-Cut Theorem" is closely related to our augmentable path theorem; however, unlike that theorem, it does not lead immediately to an algorithm for finding the maximum flow. Instead, its importance is primarily in the theoretical aspects of the subject. It is an example of a "duality" theorem, many of which are related to one another. If you are familiar with linear programming, you might like to know that this duality theorem can be proved from the linear programming duality theorem. We need a definition.

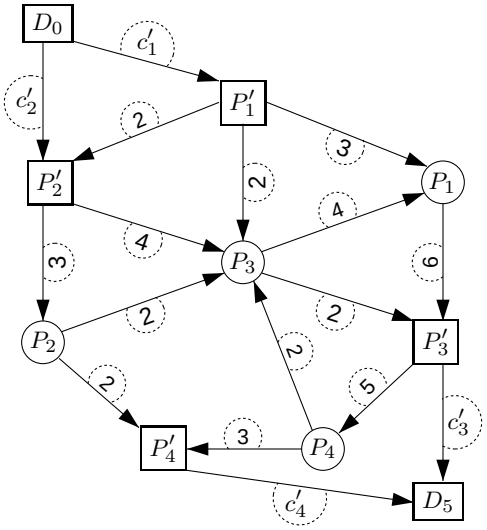


Figure 6.9 New depots for Exercise 6.4.5.

Definition 6.8 Cut partition *Given $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$, any ordered partition (A, B) of V with $\mathcal{D}_{\text{in}} \subseteq A$ and $\mathcal{D}_{\text{out}} \subseteq B$ will be called a* **cut partition**. *A* **cut set** *is a subset F of the edges of G such that every directed path from $\mathcal{D}_{\text{in}}$ to $\mathcal{D}_{\text{out}}$ contains an edge of F. If F is a set of edges in G, the sum of $c(e)$ over all $e \in F$ is called the* **capacity** *of F. If (A, B) is a cut partition, we write $c(A, B)$ instead of $c(\text{FROM}(A, B))$ and call it the capacity of the cut partition.*

The following lemma shows that cut partitions and cut sets are closely related.

Lemma *If (A, B) is a cut partition, $\text{FROM}(A, B)$ is a cut set. Conversely, if F is a cut set, then there is a cut partition (A, B) with $\text{FROM}(A, B) \subseteq F$.*

Proof: This is left as an exercise. □

Theorem 6.8 Max-Flow Min-Cut Theorem *Let f be any flow and (A, B) any cut partition for $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$. Then*

$$\text{value}(f) \leq c(A, B)$$

and, if f is a maximum flow, then there is a cut partition (A, B) such that $\text{value}(f) = c(A, B)$. The results are valid if we replace the cut partition (A, B) with the cut set F.

Proof: The inequality $\text{value}(f) \leq c(A, B)$ follows immediately from (6.11) and the fact that f takes on only nonnegative values. Suppose that f is a maximum flow. Let $A = \mathcal{A}(f)$ (and, therefore, $B = V - \mathcal{A}(f)$). It follows from the claim following (6.11) that $\text{value}(f) = c(A, B)$. To change from cut partition to cut set, apply the lemma. □

Why is this called the Max-Flow Min-Cut Theorem? The inequality $\text{value}(f) \leq c(A, B)$ implies that the maximum $\text{value}(f)$ over all possible admissible flows f is less than or equal to minimum value of $c(A, B)$ over all possible cut partitions (A, B). The fact that equality holds for maximum flows and certain cut partitions says that "The maximum value over all flows is equal to the minimum capacity over all cut partitions."

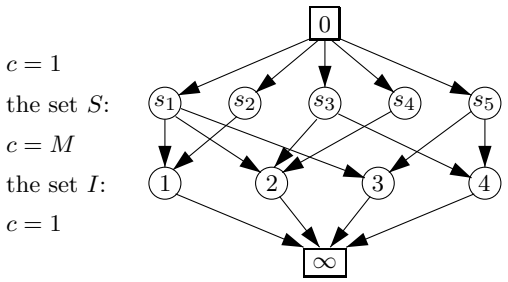


Figure 6.10 The network for $n = 4$, $A_1 = \{s_1, s_2\}$, $A_2 = \{s_1, s_3, s_4\}$, $A_3 = \{s_1, s_5\}$ and $A_4 = \{s_3, s_5\}$. Capacities appear on the left side.

Example 6.11 Systems of distinct representatives Let S be a finite set and suppose that $A_i \subseteq S$ for $1 \leq i \leq n$. A list $a_1, \ldots, a_n$ is called a system of representatives for the A_i's if $a_i \in A_i$ for all i. If the a_i's are distinct, we call the list a *system of distinct representatives* for the A_i's.

Systems of distinct representatives are useful in a variety of situations. A classical example is the *marriage problem*: There are n tasks and a set S of resources (e.g., employees, computers, delivery trucks). Each resource can be used to carry out some of the tasks; however, we must devote one resource to each task. If A_i is the set of resources that could carry out the ith task, then a system of distinct representatives is an assignment of resources to tasks. An extension of this, which we will not study, assigns a value for each resource-task pair. A higher value means a greater return from the resource-task pairing due to increased speed, capability, or whatever. The *assignment problem* asks for an assignment that maximizes the sum of the values.

We will use the Max-Flow Min-Cut Theorem to prove the following result, which is also called the Philip Hall Theorem and the SDR Theorem.

Theorem 6.9 Marriage Theorem *With the notation given above, a system of distinct representatives (SDR) exists if and only if*

$$\left| \bigcup_{i \in I} A_i \right| \geq |I| \tag{6.12}$$

for every subset of indices $I \subseteq \underline{n}$. In other words, every collection of A_i's contains at least as many distinct a_j's as there are A_i's in the collection.

Proof: By renaming the elements of S if necessary, we may assume that S contains no integers or ∞. Let G be the simple digraph with $V = S \cup \underline{n} \cup \{0, \infty\}$ and edges of three kinds:

- $(0, s)$ for all $s \in S$;
- (i, ∞) for all $i \in \underline{n}$;
- (s, i) for all $s \in S$ and $i \in \underline{n}$ such that $s \in A_i$.

Let all edges of the form (s, i) have capacity M, a *very large* integer and let all other edges have capacity 1. Let $\mathcal{D}_{\text{in}} = \{0\}$ and $\mathcal{D}_{\text{out}} = \{\infty\}$. Such a network is shown in Figure 6.10.

Consider a flow f which is integer valued. Since a vertex $s \in S$ has one edge directed in and that edge has capacity 1, the flow out of s cannot exceed 1. Similarly, since a vertex $i \in \underline{n}$ has one edge directed out and that edge has capacity 1, the flow into i cannot exceed 1. It also follows that f takes on only the values 0 and 1.

We can interpret the edges that have $f(e) = 1$:

- $f(0, s) = 1$ for $s \in S$ means s is used as a representative;
- $f(i, \infty) = 1$ for $i \in \underline{n}$ means A_i has a representative;

- $f(s,i) = 1$ for $s \in S$ and $i \in \underline{n}$ means s is the representative of A_i.

You should convince yourself that this interpretation provides a bijection between integer valued flows f and systems of distinct representatives for *some* of the A_i's, viz., those for which $f(i,\infty) = 1$. To do this, note that for a given $s \in S$, $f(s,i) = 1$ for at most one $i \in \underline{n}$ because the flow into s is at most 1. Experiment with integer flows and partial systems of distinct representatives in Figure 6.10 to clarify this.

From the observation in the previous paragraph, value(f) is the number of sets for which distinct representatives have been found. Thus a system of distinct representatives is associated with a flow f with value$(f) = n$. If we can understand what a minimum capacity cut set looks like, we may be able to use the Max-Flow Min-Cut Theorem to complete the proof.

What can we say about a minimum capacity cut set F? Note that F contains no edges of the form (s,i) because of their large capacity. Thus $c(e) = 1$ for all $e \in F$ and so $c(F) = |F|$. Consequently, we are concerned with the minimum of $|F|$ over all cut sets F containing no edges of the form (s,i). Thus F contains edges of the form $(0,s)$ and/or (i,∞).

Let I be those $i \in \underline{n}$ such that $(i,\infty) \notin F$. What edges of the form $(0,s)$ are needed to form a cut set? If $i \in \underline{n}$ and $s \in A_i$, then we must have an edge from the path $0, s, i, \infty$ in the cut set. Thus, $i \in I$ implies that $(0,s) \in F$. It follows that $(0,s) \in F$ for every $s \in \bigcup_{i \in I} A_i$.

This is enough to form a cut set: Suppose $0, s, i, \infty$ is a path. If $i \notin I$, then $(i,\infty) \in F$. If $i \in I$, then $(0,s) \in \bigcup_{i \in I} A_i \subseteq F$.

What is $|F|$ in this case? (Figure 6.10 may help make the following discussion clearer.) We have $n - |I|$ edges of the form (i,∞) and $\left|\bigcup_{i \in I} A_i\right|$ edges of the form $(0,s)$. Thus $|F|$ is the sum of these and so the minimum capacity is

$$\min_{I \subseteq \underline{n}} \left\{ \left| \bigcup_{i \in I} A_i \right| + n - |I| \right\} \;=\; n + \min_{I \subseteq \underline{n}} \left\{ \left| \bigcup_{i \in I} A_i \right| - |I| \right\}. \tag{6.13}$$

By the Max-flow Min-cut Theorem, a system of distinct representatives will exist if and only if this is at least n. Consequently, the expression in the right hand set of braces of (6.13) must be nonnegative for all $I \subseteq \underline{n}$. □

Exercises

6.4.6. Prove the lemma about cut partitions.

*6.4.7. Prove that for a given max-flow problem, $\mathcal{A}(f)$ is the same for all maximum flows f.

6.4.8. For $r \leq n$, and $r \times n$ *Latin rectangle* is an $r \times n$ array in which each row is a permutation of $\underline{n}$ and each column contains no element more than once. If $r = n$, each column must therefore be a permutation of $\underline{n}$. Such a configuration is called a Latin square. The goal of exercise is to prove that it is always possible to add $n - r$ rows to such an $r \times n$ Latin rectangle to obtain a Latin square.

(a) Suppose we are given and $r \times n$ Latin rectangle L with $r < n$. In the notation for systems of distinct representatives, let $S = \underline{n}$ and let A_i be those elements of S that do not appear in the ith column of L. Prove that a system of distinct representatives could be appended to L to obtain and $(r+1) \times n$ Latin rectangle.

(b) Prove that each $s \in S$ appears in exactly $n - r$ of the A_i's.

(c) Use the previous result and $|A_i| = n - r$ to prove that $|A_I| \geq |I|$ and so conclude that a system of distinct representatives exists.

(d) Use induction on $n - r$ to prove that an $r \times n$ Latin rectangle can be "completed" to a Latin square.

6.4.9. The purpose of this exercise is to prove the Marriage Theorem without using flows in networks. The proof will be by induction on n.

(a) Prove the theorem for $n = 1$.

(b) For the induction step, consider two cases, either (6.12) is strict for all $I \neq \underline{n}$ or it is not. Prove the induction step in the case of strictness by proving that we may choose any $a \in A_n$ as the representative of A_n.

(c) Suppose that equality holds in (6.12) for $I \neq \underline{n}$. Let $X = \cup_{i \in I} A_i$ and $B_i = A_i - X$, the set of those elements of A_i which are not in X. Prove that

$$\left| \bigcup_{i \in R} B_i \right| \geq |R|$$

for all $R \subseteq (\underline{n} - X)$. Use the induction hypothesis twice to obtain a system of distinct representatives.

*6.4.10. Let G be a directed graph and let u and v be two distinct vertices in G. Suppose that (u, v) is not an edge of G. A set of directed paths from u to v in G is called "edge disjoint" if none of the paths share an edge. A set F of edges of G is called an "edge cutset" for u and v if every directed path from u to v in G contains an edge in F. Prove that the cardinality of the largest set of edge disjoint paths equals the cardinality of the smallest edge cutset.
Hint. Make a network of G with source u and sink v.

*6.4.11. State and prove a result like that in Exercise 6.4.10 for graphs.

*6.4.12. Using the idea in Exercise 6.4.4 state and prove results like the two previous exercises with "edge" replaced by "vertex other that u and v" in the definitions of disjoint and cutset. The undirected result is called *Menger's theorem.*

*6.5 Probability and Simple Graphs

Probability theory is used in two different ways in combinatorics.

It can be used to show that, if something is large enough, then it must have some property. For example, it was shown in Example 1.25 (p. 29) that certain error correcting codes must exist if the code words were long enough. Estimates obtained this way are often quite far from best possible. On the other hand, better estimates may be hard to find.

Probability theory is also used to study how random objects behave. For example, what is the probability that a "random graph" with n vertices and $n - 1$ edges is a tree?

What do we mean by random graphs? It may be more instructive to ask "What are some questions asked about random graphs?" Here are some examples, sometimes a bit vaguely stated. You should think of n as large and the graphs as simple.

- What is the probability that a random n-vertex, $(n - 1)$-edge graph is a tree?
- How many edges must an n-vertex random graph have so that it is likely to be connected?
- On average, how many leaves does an n-vertex tree possess and how far is a random tree likely to differ from this average?
- How many colors are we likely to need to color a random n-vertex graph that has kn-edges?
- How can we generate graphs at random so that they resemble the graph of connections of computers on the internet?

To answer questions like these, we must be clear on what is meant by "random" and "likely". Sometimes this can get rather technical; however, there are simple examples.

To begin, we usually consider a set S_n of (simple) graphs with n-vertices and make it into a probability space. An obvious way to do this is with the uniform probability. For example, let S_n be the set of $(n-1)$-edge simple graphs with vertex set $\underline{n}$ and let Pr be the uniform distribution. What is the probability that a random such graph is a tree? By Exercise 5.1.2 (p. 124), there are $\binom{N}{k}$ k-edge graphs, where $N = \binom{n}{2}$. Since a tree has $n-1$ edges, we want $k = n-1$. By Example 5.10 (p. 143), there are n^{n-2} trees. Thus the answer is $n^{n-2} \big/ \binom{N}{n-1}$.

Example 6.12 Graphs with few edges Suppose a graph has few edges. What are some properties we can expect to see?

To answer such questions, we need a probability space. Let $\mathcal{G}(n,k)$ be the probability space gotten by taking the uniform probability on the set of k-edge simple graphs with vertex set $\underline{n}$. For convenience, let $N = \binom{n}{2}$.

If our graph has $n-1$ edges, it could be a tree; however, we'll show that this is a rare event. (You were asked to estimate this probability in Exercise 5.5.15(b). We'll do it here.) The number of such graphs is

$$\begin{aligned}
\binom{N}{n-1} &= \frac{N(N-1)\cdots(N-(n-1)+1)}{(n-1)!} > \frac{\big(N-(n-1)\big)^{n-1}}{(n-1)!} \\
&= \frac{\big((n-2)(n-1)/2\big)^{n-1}}{(n-1)!} \\
&\sim \frac{\big((n-2)(n-1)/2\big)^{n-1}}{\sqrt{2\pi(n-1)}\big((n-1)/e\big)^{n-1}} && \text{by Stirling's formula} \\
&> \frac{\big(e(n-2)/2\big)^{n-1}}{\sqrt{2\pi n}}.
\end{aligned}$$

Since there are n^{n-2} trees, for large n the probability that a random graph in $\mathcal{G}(n,n-1)$ is a tree is less than

$$\frac{n^{n-2}\sqrt{2\pi n}}{\big(e(n-2)/2\big)^{n-1}} = \sqrt{2\pi/n}\,(2/e)^{n-1}\left(1+\frac{2}{n-2}\right)^{n-1}. \tag{6.14}$$

From calculus, $\lim_{x\to 0}(1+x)^{a/x} = e^a$. With $x = \frac{2}{n-2}$ and $a = 2$, we have $\left(1+\frac{2}{n-2}\right)^{n-2} \sim e^2$. Since $2/e < 1$, (6.14) goes to zero rapidly as n gets large. Thus trees are rare.

You should be able to show that a simple graph with n vertices and $n-1$ edges that is not a tree is not connected and has cycles. That leads naturally to two questions:

- How large must k be so that most graphs in $\mathcal{G}(n,k)$ have cycles?
- How large must k be so that most graphs in $\mathcal{G}(n,k)$ are connected?

The first question will be looked at some in the exercises. We'll look at the second question a bit later. □

Of course, the uniform distribution is not the only possible distribution, but why would anyone want to choose a different one? Suppose we are studying graphs with n vertices and k edges. The fact that the number of edges is fixed can be awkward. For example, suppose u, v and w are three distinct vertices. If we know whether or not $\{u, v\}$ is an edge, this information will affect the probability that $\{u, w\}$ is an edge:

- If $\{u, v\}$ is not an edge, there must be k edges among the remaining $N - 1$ possible edges, so the probability that $\{u, w\}$ is an edge is equal to $\frac{k}{N-1}$.
- If $\{u, v\}$ is an edge, there must be $k - 1$ edges among the remaining $N - 1$ possible edges, so the probability that $\{u, w\}$ is an edge is equal to $\frac{k-1}{N-1}$.

Here is a way to avoid that. For each $e \in \mathcal{P}_2(\underline{n})$, make the set $\{\text{“}e \in G\text{”}, \text{“}e \notin G\text{”}\}$ into a probability space by setting $\Pr(e \in G) = p$ and $\Pr(e \notin G) = 1 - p$. We can think of “$e \in G$” as the event in which e is in a randomly chosen graph. Let $\mathcal{G}_p(n)$ be the product of these probability spaces over all $e \in \mathcal{P}_2(\underline{n})$. Let $X(G)$ be the number of edges in G. It can be written as a sum of the N independent random variables

$$X_e(G) \;=\; \begin{cases} 1 & \text{if } e \in G, \\ 0 & \text{if } e \notin G. \end{cases} \tag{6.15}$$

By independence, $\mathbf{E}(X) = pN$ and $\mathbf{var}(X) = Np(1 - p)$ because $p(1 - p)$ is the variance of a (0,1)-valued random variable whose probability of being 1 is p. With $p = k/N$, we expect to see k edges with variance $kp(1 - p) < k$. By Chebyshev's inequality (C.3) (p. 385)

$$\Pr(|X - k| > Ck^{1/2}) \;<\; 1/C^2.$$

Thus, with $C = k^{1/3}$, dividing $|X - k| > k^{1/6}$ by k, and using $\Pr(A') = 1 - \Pr(A)$, we have

$$\Pr\left(\frac{|X - k|}{k} \le k^{-1/6}\right) \;>\; 1 - k^{-2/3}. \tag{6.16}$$

Since $(|X-k|/k) \times 100$ is the percentage deviation of the X from k, (6.16) tells us that this deviation is very likely to be small when k is large.[3]

Because a random graph in $\mathcal{G}_p(n)$ has very nearly pN edges, results for $\mathcal{G}_p(n)$ with $p = k/N$ almost always hold for $\mathcal{G}(n, k)$ as well. Since $\mathcal{G}_p(n)$ is usually easier to study than $\mathcal{G}(n, k)$, people often study it.

If we want to consider all graphs with vertex set $\underline{n}$ with each of the 2^N graphs equally likely, we simply study $\mathcal{G}_{1/2}(n)$ because any particular graph with q edges occurs with probability

$$(1/2)^q(1 - 1/2)^{N-q} \;=\; (1/2)^N \;=\; 2^{-N},$$

a value that is the same for all n-vertex graphs.

[3] For those familiar with the normal approximation to the binomial distribution, the number of edges is binomially distributed. Using this, one can avoid using Chebyshev's inequality and derive stronger results than (6.16).

Example 6.13 The clique number of a random graph A *clique* in a graph G is a subgraph H such that every pair of vertices of H is connected by an edge. The size of the clique is the number of vertices of H. The *clique number* of a graph is the size of its largest clique. A k-vertex clique is called a k-clique.

What can we say about the clique number of a random graph; that is, a graph chosen using $\mathcal{G}_{1/2}(n)$? We'll get an upper bound on this number for most graphs.

Notice that if a graph contains a K-clique, then it contains an k-clique for all $k \le K$. Thus if a graph does not contain a k-clique, its clique number must be less than k. If we can show that most n-vertex graphs do not have a k-clique, it will follow that the clique number of most n-vertex graphs is less than k.

We'll begin by looking at the expected number of k-cliques in an n-vertex graph. When $W \subseteq \underline{n}$, let X_W be 1 if the vertices W form a clique and 0 otherwise. The probability that $X_W = 1$ is $(1/2)^{\binom{|W|}{2}}$ since there are $\binom{|W|}{2}$ pairs of vertices in W, each of which must be connected by an edge to form a clique. Edges that do not connect two vertices in W don't matter—they can be present or absent. Summing over all k-element subsets of $\underline{n}$, we have

$$\begin{aligned}
\mathbf{E}(\text{number of } k\text{-cliques}) &= \mathbf{E}\Bigg(\sum_{\substack{W\subset\underline{n}\\|W|=k}} X_W\Bigg) \\
&= \sum_{\substack{W\subset\underline{n}\\|W|=k}} \mathbf{E}(X_W) \;=\; \sum_{\substack{W\subset\underline{n}\\|W|=k}} \Pr(X_W = 1) \\
&= \sum_{\substack{W\subset\underline{n}\\|W|=k}} (1/2)^{\binom{k}{2}} \;=\; \binom{n}{k} 2^{-\binom{k}{2}}.
\end{aligned}$$

Since the number of k-cliques in a graph is a nonnegative integer,

$$\begin{aligned}
\Pr(\text{at least one } k\text{-clique}) &= \sum_{j>0} \Pr(\text{exactly } j \text{ } k\text{-cliques}) \\
&\le \sum_{j\ge 0} j \Pr(\text{exactly } j \text{ } k\text{-cliques}) \;=\; \mathbf{E}(\text{number of } k\text{-cliques}) \\
&= \binom{n}{k} 2^{-\binom{k}{2}} \;\le\; \frac{n^k}{k!} 2^{-\binom{k}{2}} \;=\; \frac{2^{k(\log_2 n - k/2)}}{2^{-k/2}k!}.
\end{aligned}$$

Since $2^{-k/2}k!$ is large when k is large, the probability of a k-clique will be small when $k/2 \ge \log_2 n$. Thus almost all graphs have clique number less than $2\log_2 n$.

What can be said in the other direction? A lower bound is given in the Exercise 6.5.6. □

Example 6.14 Triangles in random graphs Using $\mathcal{G}_p(n)$ for our probability space, we want to look at the number of triangles in a random graph. For $u, v, w \in \underline{n}$, let

$$X_{u,v,w} \;=\; \begin{cases} 1 & \text{if the edges } \{u,v\},\ \{u,w\},\ \{v,w\} \text{ are present,} \\ 0 & \text{otherwise.} \end{cases}$$

We claim $\Pr(X_{u,v,w} = 1) \;=\; p^3$. How can we see this? Intuitively, each edge has probability p of being present and they are independent, so we get p^3. More formally, since our probability space is a product space,

$$X_{u,v,w} \;=\; X_{\{u,v\}} X_{\{u,w\}} X_{\{v,w\}} \;=\; p^3,$$

where the $X_{x,y}$ are the random variable defined in (6.15).

Thus the expected value of $X_{u,v,w}$ is p^3. Since there are $\binom{n}{3}$ choices for $\{u,v,w\}$, the expected number of triangles in $\binom{n}{3}p^3$.

Let's compute the variance in the number of triangles. We have to be careful: Triangles are not independent because they may share edges. The safest approach is to define a random variable T that equals the number of triangles:

$$T \;=\; \sum_{t\in\mathcal{P}_3(\underline{n})} X_t, \quad \text{where} \quad X_{\{u,v,w\}} \;=\; X_{u,v,w}.$$

Since $\mathbf{var}(T) \;=\; \mathbf{E}(T^2) - \mathbf{E}(T)^2$ and we know $\mathbf{E}(T) \;=\; \binom{n}{3}p^3$, we need to compute $\mathbf{E}(T^2)$. It equals $\sum \mathbf{E}(X_sX_t)$, the sum ranging over all $s,t \in \mathcal{P}_3(\underline{n})$. There are three cases to consider:

- $s = t$ There are $\binom{n}{3}$ terms like this and $\mathbf{E}(X_sX_t) \;=\; \mathbf{E}(X_s^2) \;=\; \mathbf{E}(X_s) \;=\; p^3$.
- $|s\cap t| = 2$ There are $\binom{n}{3}\binom{3}{2}\binom{n-3}{1}$ terms like this since we have $\binom{n}{3}$ choices for s, $\binom{3}{2}$ ways to select two elements of s to include in t, and $\binom{n-3}{1}$ ways to complete t. Since there are a total of five edges in the two triangles, $\mathbf{E}(X_sX_t) \;=\; p^5$.
- $|s\cap t| < 2$ Since there are a total of $\binom{n}{3}^2$ terms in the sum $\sum \mathbf{E}(X_sX_t)$ and we have dealt with $\binom{n}{3} + \binom{n}{3}3(n-3)$ terms already, there are

$$\binom{n}{3}^2 - \binom{n}{3} - \binom{n}{3}3(n-3)$$

remaining. Since there are six edges in the two triangles, $\mathbf{E}(X_sX_t) = p^6$. Putting all this together

$$\begin{aligned}\mathbf{var}(T) \;&=\; \Big(\binom{n}{3}^2 - \binom{n}{3} - \binom{n}{3}3(n-3)\Big)p^6 + \binom{n}{3}3(n-3)p^5 + \binom{n}{3}p^3 - \left(\binom{n}{3}p^3\right)^2\\ &=\; \binom{n}{3}3(n-3)p^5(1-p) + \binom{n}{3}p^3(1-p^3).\end{aligned}$$

Now we want to use Chebyshev's inequality to find out when most graphs have at least one triangle. Chebyshev's inequality (C.3) (p. 385) is

$$\Pr\Big(|T - \mathbf{E}(T)| > t\sqrt{\mathbf{var}(T)}\Big) \;<\; 1/t^2.$$

If $T = 0$, then $|T - \mathbf{E}(T)| \;=\; \mathbf{E}(T) \;>\; \mathbf{E}(T) - 1$. If we set $t\sqrt{\mathbf{var}(T)} \;=\; \mathbf{E}(T) - 1$, Chebyshev's inequality tells us the probability that there is no triangle is $1/t^2$, a number that we want to be small. Solving $t\sqrt{\mathbf{var}(T)} \;=\; \mathbf{E}(T) - 1$ and putting it all together with our previous calculations:

$$\Pr(T{=}0) \;<\; \frac{\mathbf{var}(T)}{(\mathbf{E}(T)-1)^2} \;=\; \frac{\binom{n}{3}3(n-3)p^5(1-p) + \binom{n}{3}p^3(1-p^3)}{\left(\binom{n}{2}p^3 - 1\right)^2}.$$

When p is such that p is small and $\binom{n}{3}p^3$ is large, we have $\binom{n}{3}p^3 \;\approx\; n^3p^3/6$ and so we are assuming that $L = np$ is large. Using this,

$$\frac{\mathbf{var}(T)}{(\mathbf{E}(T)-1)^2} \;\approx\; \frac{n^4p^5/2 + n^3p^3/6}{n^6p^6/36} \;=\; 6\frac{3Lp+1}{L^3} \;=\; 18p/L^2 + 6/L^3,$$

We've shown that, if np is large and p is small, then a random graph in $\mathcal{G}_p(n)$ almost certainly contains a triangle. If we let p be larger, then edges are more likely and so triangles are more likely. This we don't need "p is small." In summary, If np is large then a random graph in $\mathcal{G}_p(n)$ almost certainly contains a triangle. □

Example 6.15 Growing random graphs Imagine starting out with vertices $V = \underline{n}$ and then growing a simple graph by randomly adding edges one by one. We'll describe the probable growth of such a graph.

In the first stage, we have a lot of isolated vertices (vertices on no edges) and pairs of vertices joined by an edge. As time goes by (more edges added), the single edges join up forming lots of small trees which continue to grow. Next the trees start developing cycles and so are no longer trees. Of course there are still a lot of isolated vertices and small trees around. Suddenly a threshold is passed and, in the blink of an eye, we have one large (connected) component and lots of smaller components, most of which are trees and isolated vertices. The large component starts "swallowing" the smaller ones, preferring to swallow the larger ones first. Finally, all that is left outside the large component is a few isolated vertices which are swallowed one by one and so the graph is connected. Growth continues beyond this point, but we'll stop here.

When does the graph get connected?

We can't answer this question; however, we can easily compute the expected number of isolated vertices in a random graph. When this number is near zero, we expect most random graphs to be connected. When it is not near zero, we expect a significant number of random graphs to still contain isolated vertices. We'll study this with the $\mathcal{G}_p(n)$ model. This isn't quite the correct thing to do since we're adding edges one by one. However, it's harder to use $\mathcal{G}(n,k)$ and we're not planning on proving anything—we just want to get an idea of what's true.

In $\mathcal{G}_p(n)$, a vertex v will be isolated if none of the possible $n-1$ edges connecting it to the rest of the graph are present. Thus the probability that v is isolated is $(1-p)^{n-1}$. Since there are n vertices, the expected number of isolated vertices is $n(1-p)^{n-1}$. When p is small, $1-p \approx e^{-p}$ and so the expected number of isolated vertices is about $ne^{-p(n-1)} = e^{\ln n - p(n-1)}$. This number will be near zero if $p(n-1) - \ln n$ is a large positive number. This is the same as $pn - \ln n$ being large and positive. In this case, isolated vertices are unlikely. In other words, they've all probably been swallowed by the big component and the graph is connected. On the other hand, if $pn - \ln n$ is large and negative, $e^{\ln n - p(n-1)}$ will be large. In other words, we expect a lot of isolated vertices. Thus $p \approx \frac{\ln n}{n}$ is the critical point when a graph becomes connected. Since we expect about $\binom{n}{2}p \approx \frac{n \ln n}{2}$ edges, a graph should become connected when it has around $\frac{n \ln n}{2}$ edges. □

So far we've studied random graphs and asked what can be expected. Now we'll look at a different problem: we want to *guarantee* that something must happen in a graph.

Example 6.16 Bipartite subgraphs A graph (V', E') is *bipartite* if its vertices can be partitioned into two sets V_1' and $V_2' = V' - V_1'$ such that the edges of the graph only connect vertices in V_1' and V_2'. In other words, if $\{x, y\} \in E'$, then one of x, y is in V_1' and the other is in V_2'.

Given a graph $G = (V, E)$ with n vertices and k edges, we want to find a bipartite subgraph with as many edges as possible. How many edges can we guarantee being able to find? Since we want as many edges a possible, we may as well use all the vertices in G. Thus our bipartite graph will be (V, E') where V is partitioned into V_1' and V_2' and $E' \subseteq E$ are those edges which connect a vertex in V_1' to a vertex in V_2'. Thus, our partition V_1', V_2' determines E'.

Since the example is in this section, you can tell we're going to use probability somehow. But how? (Our previous methods won't work since we are given a particular graph G.) We can choose the partition of V randomly.

Our probability space will be the uniform probability on the set of all subsets of V. A subset will be V_1' and its complement $V - V_1'$ will be V_2'. For every edge $e = \{x, y\} \in E$, define a random variable X_e by

$$X_e(V_1') = \begin{cases} 0, & \text{if both ends of } e \text{ are in } V_1' \text{ or in } V_2', \\ 1, & \text{if one end of } e \text{ is in } V_1' \text{ and the other in } V_2'. \end{cases}$$

You should be able to see that the number of edges in the bipartite graph is $X(V_1')$, which we define by

$$X(V_1') \;=\; \sum_{e\in E} X_e(V_1').$$

We can think of the probability space as a product space as follows. For each $v \in V$, choose it with probability $1/2$, independent of the choices made on the other vertices. The chosen vertices form V_1'. The probability of choosing V_1' is

$$(1/2)^{|V_1'|}(1/2)^{|V-V_1'|} \;=\; (1/2)^{|V|},$$

where $(1/2)^{|V_1'|}$ is the probability of choosing each of the vertices in V_1' and $(1/2)^{|V-V_1'|}$ is the probability of *not* choosing each of the vertices in $V - V_1'$. Thus, this probability space gives equal probability to every subset of V, just like our original space. Consequently, we can carry out calculations in either our original probability space or the product space we just introduced. Since the product space has a lot of independence built into it, calculation here is often easier. In particular, $\Pr(X_e = 1) = 1/2$ for any edge $\{u, v\}$ since $X_e = 1$ if and only if we make the same decisions for u and v (both included or both excluded) and this has probability $(1/2)^2 + (1/2)^2 \;=\; 1/2$. Thus $\mathbf{E}(X_e) \;=\; 1 \times (1/2) + 0 \times (1/2) \;=\; 1/2$.

Since a random variable must sometimes be at least as large as it's average, there must be a $V_1' \subseteq V$ with $X(V_1') \geq \mathbf{E}(X)$. Thus there is a bipartite subgraph with at least $\mathbf{E}(X)$ edges. Since $\mathbf{E}(X_e) \;=\; 1/2$, $\mathbf{E}(X) \;=\; k/2$. Since the expected number of edges in a randomly constructed bipartite subgraph is $k/2$, at least one of these subgraphs has at least $k/2$ edges. In other words, there is a bipartite subgraph containing at least half the edges of G. □

Exercises

Remember the following!

- Expectation is linear: $\mathbf{E}(X_1 + \cdots + X_k) \;=\; \mathbf{E}(X_1) + \cdots + \mathbf{E}(X_k)$.
- $\Pr(A_1 \cap \cdots \cap A_m) \;\leq\; \Pr(A_1) + \cdots + \Pr(A_m)$, especially when $\Pr(A_1) \;= \cdots =\; \Pr(A_m)$.

6.5.1. Compute the following for a random graph in $\mathcal{G}_p(n)$.

(a) The expected number of vertices of degree d.
Hint. Let $X_v = 1$ if v has degree d and $X_v = 0$ otherwise. Study $\sum X_v$.

(b) The expected number of 4-cycles.

(c) The expected number of induced 4-cycles. (An induced subgraph of a graph G is a subset of vertices together with *all* the edges in G that connect the vertices.)

6.5.2. An *embedding* of a simple graph $H = (V_H, E_H)$ into a simple graph $G = (V_G, E_G)$ is an injection $\varphi : V_H \to V_G$ such that $\varphi(E_H) \subseteq E_G$, where we define $\varphi\{u, v\} = \{\varphi(u), \varphi(v)\}$. If $\varphi(E_H) = E_G \cap \mathcal{P}_2(V_H)$, we call the embedding *induced*.

(a) Prove that the expected number of embeddings of H in a random graph in $\mathcal{G}_p(n)$ when $n \geq |V_H|$ is

$$n(n-1)\cdots(n-|V_H|+1)p^{|E_H|} \;=\; \frac{n!\; p^{|E_H|}}{(n-|V_H|)!}.$$

(b) Repeat (a) for induced embeddings.

(c) In Example 6.14, we showed that the expected number of triangles in a random graph is $\binom{n}{3}p^3$. If part (a) of the present exercise is applied when H is a triangle, we obtain $6\binom{n}{3}p^3$ for the expected number of embeddings. Explain the difference.

6.5.3. In this exercise we'll find a bound on k (as a function of n) so that most graphs in $\mathcal{G}(n,k)$ do not have cycles. Since we know most graphs in $\mathcal{G}(n,n-1)$ have cycles, we'll assume $k \le n-1$. For $C \subseteq \underline{n}$, let $\mathcal{G}_C$ be those graphs in which C is a cycle. As usual, $N = \binom{n}{2}$.

(a) Show that

$$\Pr(G \text{ has a cycle}) \;\le\; \sum \Pr(\mathcal{G}_C),$$

where the sum is over all subsets of $\underline{n}$ that contain at least three elements.

(b) By arranging the vertices in C in a cycle and then inserting the remaining edges, prove that

$$\Pr(\mathcal{G}_C) \;=\; \frac{(c-1)!/2\,\binom{N-c}{k-c}}{\binom{N}{k}} \quad \text{where } c = |C|.$$

Remember that, unlike cycles in permutations, cycles in graphs do not have a direction.

(c) Conclude that

$$\Pr(G \text{ has a cycle}) \;\le\; \sum_{c=3}^{k} \binom{n}{c} \frac{(c-1)!/2\,\binom{N-c}{k-c}}{\binom{N}{k}}.$$

(d) Show that the term c in the previous sum equals

$$\frac{k!}{2c\,(k-c)!} \prod_{i=0}^{c-1} \frac{n-i}{N-i} \;<\; \frac{k^c}{2c} \left(\frac{n}{N}\right)^c.$$

6.5.4. We'll redo the previous exercise using $\mathcal{G}_p(n)$.

(a) Let $v_1, \ldots, v_k$ be a list of vertices. Compute the probability that $v_1, \ldots, v_k, v_1$ is a cycle

(b) Show that the probability that a random graph in $\mathcal{G}_p(n)$ contains a k-cycle is less than $n^k p^k$.

(c) Show that the probability that a random graph in $\mathcal{G}_p(n)$ has a cycle is less than $(pn)^3$ when $pn < 2/3$.

6.5.5. This exercise relates to Example 6.16.

(a) A simple graph $G = (V,E)$ is *complete* if it contains all possible edges; that is, $E = \mathcal{P}_2(V)$. Prove that, if $|V| = 2n$, we can construct a bipartite subgraph with n^2 edges. Obtain a similar result if $|V| = 2n+1$.

(b) How close is this result to the lower bound in the example?

(c) Prove that when $|V|$ and k are large, there is a graph $G = (V,E)$ such that $|E| = k$ and the relative error between the lower bound and the best bipartite subgraph of G is $O(1/k^{1/2})$. (Of course we need $k \le |\mathcal{P}_2(V)|$ or there will be no simple graph.)
Hint. Use a complete graph in your construction.

(d) Prove that, if $G(V,E)$ can be properly colored using three colors, then it has a bipartite subgraph with at least $2|E|/3$ edges.

6.5.6. We want to find a number k (depending on n) such that most n-vertex graphs have a k-clique. Divide the vertices in $\lfloor n/k \rfloor$ sets of size k and one (possibly empty) smaller set.

(a) Show that the probability that none of these $\lfloor n/k \rfloor$ sets is a k-clique is $(1 - 2^{-\binom{k}{2}})^{\lfloor n/k \rfloor}$.

(b) It can be shown that $1 - x < e^{-x}$ for $x > 0$. Using this and the previous part, conclude that, for some constant A, almost all n-vertex graphs have a k-clique when $k \le A(\log n)^{1/2}$.

6.6 Finite State Machines

A "finite state machine" is simply a device that can be in any one of a finite number of situations and is able to move from one situation to another. The classic example (and motivation for the subject) is the digital computer. If no peripherals are attached, then the state at any instant is what is stored in the machine. You may object that this fails to take into account what instruction the machine is executing. Not so; that information is stored temporarily in parts of the machine's central processing unit. We can expand our view by allowing input and output to obtain a finite state machine with I/O.

By formalizing the concept of a finite state machine, computer scientists hope to capture the essential features of some aspects of computing. In this section we'll study a very restricted formalization. These restricted devices are called "finite automata" or "finite state machines." The input to such machines is fed in one symbol at a time and cannot be reread by the machine.

Turing Machines

A *Turing machine*, introduced by A.M. Turing in 1937, is a more flexible concept than a finite automaton. It is equipped with an arbitrarily long tape which it can reposition, read and write. To run the machine, we write the input on a blank tape, position the tape in the machine and turn the machine on. We can think of a Turing machine as computing a function: the input is an element of the function's domain and the output is an element of the function's range, namely the value of the function at that input. The input and/or the output could be nothing. In fact, the domain of the function is any finite string of symbols, where each symbol must be from some finite alphabet; eg. $\{0, 1\}$. Of course, the input might be something the machine wasn't designed to handle, but it will still do something.

How complicated a Turing machine might we need to build? Turing proved that there exists a "universal" Turing machine $\mathcal{U}$ by showing how to construct it. If $\mathcal{U}$'s input tape contains

1. $\mathrm{D}(\mathcal{T})$, a description of any Turing machine $\mathcal{T}$ and
2. the input I for the Turing machine $\mathcal{T}$,

then $\mathcal{U}$ will produce the same output that would have been obtained by giving $\mathcal{T}$ the input I. This says that regardless of how complicated an algorithm we want to program, there is no need to build more than one Turing machine, namely the universal one $\mathcal{U}$. Of course, it might use a lot of time and a lot of tape to carry out the algorithm, so it might not be practical. Suprisingly, it can be shown that $\mathcal{U}$ will, in some sense, be almost as fast as the the Turing machine that it is mimicking. This makes it possible to introduce a machine independent measure of the complexity of a function.

Although Turing machines seem simple, it is believed that anything that can be computed by any possible computer can be computed by a Turing machine. (This is called *Church's Thesis*.) Such computable functions are called *recursive functions*. Are there any functions which are not recursive?

Examples of nonrecursive functions are not immediately obvious. Here's one. "Given a Turing machine $\mathcal{T}$ and input I, will the Turing machine eventually stop?" As phrased this isn't quite a fair function since it's not input for a Turing machine. We can change it slightly: "Given $\mathrm{D}(\mathcal{T})$ (a machine readable description of $\mathcal{T}$) and the input I, will the universal Turing machine $\mathcal{U}$ eventually stop?" For obvious reasons, this is called the *halting problem*. You may wonder why this can be thought of as a function. The domain of the function is all possible pairs D, I where D is any machine readable description of a machine and I is any possible input for a machine. The range is {"yes", "no"}. The machine $\mathcal{U}$ computes the value of the function.

Theorem 6.10 *The halting problem is nonrecursive.*

Proof: We can't give a rigorous proof here; in fact, we haven't even defined our terms precisely. Nevertheless, we can give the idea for a proof. Suppose there existed a Turing machine $\mathcal{H}$ that could solve the halting problem. Create a Turing machine $\mathcal{B}$ that contains within itself what is essentially a subroutine equivalent to $\mathcal{H}$ and acts as follows. Whatever is written on the input tape, it makes a copy of it. It then "calls" $\mathcal{H}$ to process as input the original input together with the copy. If $\mathcal{H}$ says that the answer is "Doesn't stop," then $\mathcal{B}$ stops; otherwise $\mathcal{B}$ enters an infinite loop.

How does this rather strange machine behave? Suppose $\mathcal{B}$ is given $\mathrm{D}(\mathcal{T})$ as input. It then passes to $\mathcal{H}$ the input $\mathrm{D}(\mathcal{T})$ $\mathrm{D}(\mathcal{T})$ and so $\mathcal{H}$ solves the halting problem for $\mathcal{T}$ with $\mathrm{D}(\mathcal{T})$ as input. In other words:

$$\text{If } \mathcal{B} \text{ is given } \mathrm{D}(\mathcal{T}), \text{ it halts if and only if } \mathcal{T} \text{ with input } \mathrm{D}(\mathcal{T}) \text{ would run for ever.} \qquad 6.17$$

What does $\mathcal{B}$ do with the input $\mathrm{D}(\mathcal{B})$? We simply use (6.17) with $\mathcal{T}$ equal to $\mathcal{B}$. Thus, $\mathcal{B}$ with input $\mathrm{D}(\mathcal{B})$ halts if and only if $\mathcal{B}$ with input $\mathrm{D}(\mathcal{B})$ runs for ever. Since this is self-contradictory, there is either an error in the proof, an inconsistency in mathematics, or a mistake in assuming the existence of $\mathcal{H}$. We believe that the last is the case: There cannot be a Turing machine $\mathcal{H}$ to solve the halting problem. □

We can describe the previous proof heuristically. If $\mathcal{H}$ exists, then it predicts the behavior of any Turing machine. $\mathcal{B}$ with input $\mathrm{D}(\mathcal{B})$ is designed to ask $\mathcal{H}$ how it will behave and then do the opposite of the prediction.

Finite State Machines and Digraphs

Consider a finite state machine that receives input one symbol at a time and enters a new state based on that symbol. We can represent the states of the machine by vertices in a digraph and the effect of the input i in state s by a directed edge that connects s to the new state and contains i and the associated output in its name. The following example should clarify this.

Example 6.17 Binary addition We would like to add together two nonnegative binary numbers and output the sum. The input is given as pairs of digits, one from each number, starting at the right ends (units digits) of the input. The pair 22 marks the end of the input. Thus to add 010 and 110 you would input the four pairs 00, 11, 01 and 22 in that order. In other words,

$$\text{the sum problem} \quad \begin{array}{rl} & A_nA_{n-1}\cdots A_1 \\ + & B_nB_{n-1}\cdots B_1 \\ \hline & C_{n+1}C_nC_{n-1}\cdots C_1 \end{array} \quad \text{becomes} \quad A_1B_1, \ldots, A_{n-1}B_{n-1}, A_nB_n, 22.$$

The output is given as single digits with 2 marking the end of the output, so the output for our example would be 00012. (The sum is backwards, $C_1 \ldots C_{n-1}, C_n, C_{n+1}, 2$, because the first output is the units digit.) We have two internal states: carry (C) and no carry (N) You should verify that the adder can be described by the table in Figure 6.11. The entry (o, s_2) in position (s_1, i) says that if the machine is in state s_1 and receives input i, then it will produce output o and move to state s_2. It is called the *state transition table* for the machine. Note that being in state C (carry) and receiving 22 as input causes two digits to be output, the carry digit and the termination digit 2.

We can associate a digraph (V, E, φ) with the tabular description, where $V = \{\mathrm{N}, \mathrm{C}\}$, each edge is a 4-tuple $e = (s_1, i, o, s_2)$, $\varphi(e) = (s_1, s_2)$ and i and o are the associated input and output, respectively. In drawing the picture, a shorthand is used: the label $00, 22 : 1, 12$ on the edge from C to N in Figure 6.11 stands for the two edges (C, 00, 1, N) and (C, 22, 12, N).

This example is slightly deficient. We tacitly assumed that everyone (and the machine!) somehow knew that the machine should start in state N. We should really indicate this by labeling N as the *starting state*.

	00	01	10	11	22
N	0, N	1, N	1, N	0, C	2, N
C	1, N	0, C	0, C	1, C	12, N

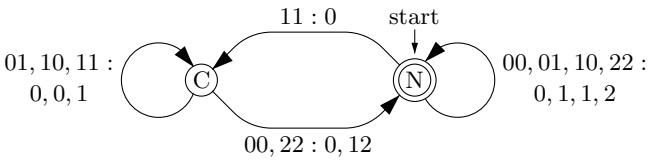


Figure 6.11 Tabular and graphical descriptions of a finite state machine for adding two binary numbers. The starting and accepting states are both N.

You can use the associated digraph to see easily what an automaton does with any given input string. Place your finger on the starting state and begin reading the input. Each time you read an input symbol, follow the directed edge that has that input symbol to whichever state (vertex) it leads and write down the output that appears on the edge. Keep this process up until you have used up all the input. □

Suppose we want a machine that simply recognizes a situation but takes no action. We could phrase this by saying that the machine either accepts (recognizes) or rejects an input string. In this case, we need not have any output; rather, we can label certain states of the machine as "accepting." This is represented pictorially by a double circle around an *accepting state*. If the machine ends up in an accepting state, then the input is accepted; otherwise, it is rejected.

Example 6.18 No adjacent ones Let's construct a machine to recognize (accept) all strings of zeroes and ones that contain no adjacent ones. The idea is simple: keep track of what you saw last and if you find a one followed by a one, reject the string. We use three states, labeled 0, 1 and R, where 0 and 1 indicate the digit just seen and R is the reject state. Both 0 and 1 are accepting states. What is the start state? We could add an extra state for this, but we get a smaller machine if we let 0 be the start state. This can be done because the string $s_1 \cdots s_n$ is acceptable if and only if $0s_1 \cdots s_n$ is. You should be able to draw a diagram of this machine. □

Example 6.19 Divisibility by five Let's construct a machine to recognize (accept) numbers which are divisible by 5. These numbers will be presented from left to right as binary numbers; i.e., starting with the highest order digits. To construct such a machine, we simply design it to carry out the usual long division algorithm that you learned many years ago. At each step in usual long division algorithm you produce a remainder to which you then append the next digit of the dividend. In effect, this multiplies the remainder by ten and adds the next digit to it. Since we are working in binary, we follow the same process but multiply by two instead of by ten. The digraph for the machine appears in Figure 6.12. No output is shown because there is none. □

Here's a formal definition of the concept of a finite automaton, which we've been using rather loosely so far.

Definition 6.9 Finite automaton *A* **finite automaton** *is a quadruple* (S, I, f, s_0) *where* S *and* I *are finite sets,* $s_0 \in S$ *and* $f\colon S \times I \to S$. *S is called the set of* **states** *of the automaton, I the set of* **input symbols** *and* s_0 *the* **starting state**. *If the automaton has* **accepting states**, *we append them to the quadruple to give a quintuple. If the automaton has a set* **output symbols** O, *then we append it to the tuple and change the definition of* f *to* $f\colon S \times I \to O \times S$. *If an input string leaves the automaton in an accepting state, we say that the automaton* **accepts** *the string or that it* **recognizes** *the string.*

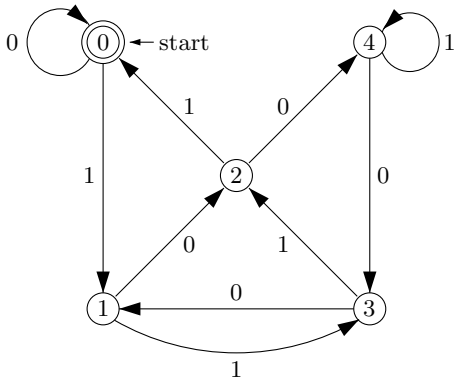


Figure 6.12 A machine to test divisibility of binary numbers by 5. The starting and accepting states are both 0. Input begins with the high order (leftmost) bit.

Example 6.20 An automaton grammar We can represent a finite automaton without output in another fashion. For each edge (s_1, i, s_2) we write $s_1 \to i, s_2$. If s_2 is an accepting state, we also write $s_1 \to i$. Suppose we begin with the starting state, say s_0 and replace it with the right side of some $s_0 \to$. If this leads to a string contains a state s, then replace s in the string with the right side of some $s \to$. After n such steps we will end up with either a string of n input symbols or a string of n input symbols followed by a state.

We claim that any string of input symbols (with no appended state) that can be produced in this fashion is accepted and conversely. Why is this true? Our replacement process mimics travelling along the digraph as dictated by the input string. We can only omit the state symbol at the end by moving to an accepting state. This is an example of a "grammar." We'll say more about grammars in Section 9.2. □

We can attempt to endow our string recognizer with "free will" by allowing random choices. At present, for each state $s \in S$ and each $i \in I$ there is precisely one $t \in S$ such that (s, i, t) is an edge of the digraph. Remove the phrase "precisely one." We can express this in terms of the function f by saying that $f\colon S \times I \to 2^S$, the subsets of S, instead of $f\colon S \times I \to S$. Here $f(s, i)$ is the set of all states t such that (s, i, t) is an edge.

Since the successor of a state is no longer uniquely defined, what happens when the machine is in state s and receives input i? If $f(s, i) = \emptyset$, the empty set, the machine stops and does not accept the string; otherwise, the machine selects by its "free will" any $t \in f(s, i)$ as its next state.

Since the outcome of a given input string is not uniquely determined, how do we define acceptance? We simply require that acceptance be possible: If it is possible for the machine to get from its starting state to an accepting state by any sequence of choices when it receives the input string X, then we say the machine accepts X. Such a machine is called a *nondeterministic finite automaton.* What we have called finite automata are often called *deterministic* finite automata.

Theorem 6.11 No "free will" *Given a nondeterministic finite automaton, there exists a (deterministic) finite automaton that accepts exactly the same input strings.*

The conclusions of this theorem are not as sweeping as our name for it suggests: We are speaking about a rather restricted class of devices and are only concerned about acceptance. The rest of this section will be devoted to the proof.

Proof: Let $\mathcal{N} = (S, I, f, s_0, A)$, with $f: S \times I \to 2^S$, be a nondeterministic finite automaton. We will construct a deterministic finite automaton $\mathcal{D} = (T, I, g, t_0, B)$ that accepts the same input strings as $\mathcal{N}$.

Let the states T of $\mathcal{D}$ be 2^S, the set of all subsets of S. The initial state of $\mathcal{D}$ will be $t_0 = \{s_0\}$ and the set of accepting states B of $\mathcal{D}$ will be the set of all those subsets of S that contain at least one element from A. We now define $g(t, i)$. Let $g(\emptyset, i) = \emptyset$. For t a nonempty subset of S, let $g(t, i)$ be the union of $f(s, i)$ over all $s \in t$; that is,

$$g(t, i) = \bigcup_{s \in t} f(s, i). \tag{6.18}$$

This completes the definition of $\mathcal{D}$.

We must prove that $\mathcal{D}$ recognizes precisely the same strings that $\mathcal{N}$ does. Let t_n be the state that $\mathcal{D}$ is in after receiving the input string $X = i_1 \dots i_n$. We claim that

$$\mathcal{N} \text{ can be in a state } s \text{ after receiving } X \text{ if and only if } s \in t_n. \tag{6.19}$$

Before proving (6.19), we'll use it to prove the theorem.

Suppose that $\mathcal{N}$ accepts X. Then it is possible for $\mathcal{N}$ to reach some accepting state $a \in A$, which is in t_n by (6.19). By the definition of B, $t_n \in B$. Thus $\mathcal{D}$ accepts X.

Now suppose that $\mathcal{D}$ accepts X. Since t_n is an accepting state of $\mathcal{D}$, it follows from the definition of B that some $a \in A$ is in t_n. By (6.19), $\mathcal{N}$ can reach a when it receives X. We have shown that (6.19) implies the theorem.

It remains to prove (6.19). We'll use induction on n. Suppose that $n = 1$. Since $t_0 = \{s_0\}$, it follows from (6.18) that $t_1 = f(t_0, i_1)$. Since this is the set of states that can be reached by $\mathcal{N}$ with input i_1, we are done for $n = 1$.

Suppose that $n > 1$. By (6.18),

$$t_n = \bigcup_{s \in t_{n-1}} f(s, i_n).$$

By the induction assumption, t_{n-1} is the set of states s that $\mathcal{N}$ can be in after receiving the input $i_1 \dots i_{n-1}$. By the definition of f, $f(s, i_n)$ is the set of states that $\mathcal{N}$ can reach from s with input i_n. Thus t_n is the set of states that $\mathcal{N}$ can be in after receiving the input $i_1 \dots i_n$. □

Exercises

6.6.1. What is the state transition table for the automaton of Example 6.19?

6.6.2. We now wish to check for divisibility by 3.

(a) Give a digraph like that in Example 6.19 for binary numbers.

(b) Give the state transition table for the previous digraph.

(c) Repeat the two previous parts when the input is in base 10 instead of base 2.

*(d) Construct an automaton that accepts decimal input, starting with the unit's digit and checks for divisibility by 3.

*(e) Construct an automaton that accepts binary input, starting with the unit's digit and checks for divisibility by 3.

6.6.3. Design a finite automaton to recognize all strings of zeroes and ones in which no maximal string of ones has even length. (A maximal string of ones is a string of adjacent ones which cannot be extended by including an adjacent element of the string.)

6.6.4. An automaton is given by $(\{b, s, d, z\},\ \{+, -, 0, 1, \ldots, 9\}, f\ , b,\ \{d\})$, where $f(b, +) = s$, $f(b, -) = s$, $f(t, k) = d$ for $t = b, s, d$ and $0 \leq k \leq 9$, and $f(t, i) = z$ for all other $(t, i) \in S \times I$.

(a) Draw the digraph for the machine.

(b) Describe the strings recognized by the machine.

(c) Describe the strings recognized by the machine if both s and d are acceptance states.

6.6.5. A "floating point number" consists of two parts. The first part consists of an optional sign followed by a nonempty string of digits in which at most one decimal point may be present. The second part is either absent or consists of "E" followed by a signed integer. Draw the digraph of a finite automaton to recognize floating point numbers.

6.6.6. A symbol i_k a string $i_i \ldots i_n$ is "isolated" if (i) either $k = 1$ or $i_k \neq i_{k-1}$, and (ii) either $k = n$ or $i_k \neq i_{k+1}$. For example, 0111010011 contains two isolated zeroes and one isolated one.

(a) Draw a digraph for a finite automaton that accepts just those strings of zeroes and ones that contain at least one isolated one.

(b) Now draw a machine that accepts strings with precisely one isolated one.

6.6.7. In this exercise you are to construct an automaton that behaves like a vending machine. To keep things simple, there are only three items costing 15, 20 and 25 cents and indicated by the inputs A, B and C, respectively. Other allowed inputs are 5, 10 and 25 cents and R (return coins). The machine will accept any amount of money up to 30 cents. Additional coins will be rejected. When input A, B or C is received and sufficient money is present, the selection is delivered and change (if any) returned. If insufficient money is present, no action is taken.

(a) Describe appropriate states and output symbols. Identify the starting state.

(b) Give the state transition table for your automaton.

(c) Draw a digraph for your automaton.

6.6.8. Suppose that $\mathcal{M} = (S, I, f, s_0, A)$ and $\mathcal{M}' = (S', I, f', s_0', A')$ are two automata with the same input symbols and with acceptance states A and A' respectively.

(a) Describe in terms of sets and functions an automaton that accepts only those strings acceptable to both $\mathcal{M}$ and $\mathcal{M}'$.
Hint. The states can be $S \times S'$.

(b) When is there an edge from (s, s') to (t, t') in your new automaton and what input is it associated with?

(c) Use this idea to describe an automaton the recognizes binary numbers which are divisible by 15 in terms of those in Example 6.19 and Exercise 6.6.2.

(d) Design a finite automaton that recognizes those binary numbers which are either divisible by 3 or divisible by 5 or both.

Notes and References

Spanning trees are one of the most important concepts in graph theory. As a result, they are discussed in practically every text. Our point of view is like that taken by Tarjan [7; Ch.6], who treats the subject in greater depth.

An extensive treatment of planarity algorithms can be found in Chapters 6 and 7 of the text by Williamson [8]. Wilson's book [98] is a readable account of the history of the four color problem.

Since flows in networks is an extremely important subject, it can be found in many texts. Papadimitriou and Steiglitz [6] and Tarjan [7] treat the subject extensively. In addition, network flows are related to linear programming, so you will often find network flows discussed in linear programming texts such as the book by Hu [4].

The marriage theorem is a combinatorial result about sets. Sperner's theorem, which we studied in Example 1.22, is another such result. For more on this subject, see the text by Anderson [1]. Related in name but not in results is the *Stable Marriage Problem*. In this case, we have two sets of the same size, say men and women. Each woman ranks all of the men and each man all of the women. The men and women are married to each other. The situation is considered stable if we cannot find a man and a woman who both rank each other higher than their mates. It can be proved that a stable marriage always exists. Gusfield and Irving [3] discuss this problem and its applications and generalizations.

The books on the probabilistic method are more advanced than the discussion here. Perhaps the gentlest book is the one by Molloy and Reed [5].

Automata are discussed in some combinatorics texts that are oriented toward computer science. There are also textbooks, such as Drobot [2] devoted to the subject.

1. Ian Anderson, *Combinatorics of Finite Sets*, Dover (2002).
2. Vladimir Drobot, *Formal Languages and Automata Theory*, W. H. Freeman (1989).
3. Dan Gusfield and Robert W. Irving, *The Stable Marriage Problem: Structure and Algorithms*, MIT Press (1989).
4. T.C. Hu, *Integer Programming and Network Flows*, Addison-Wesley (1969).
5. Michael Molloy and Bruce Reed, *Graph Coloring and the Probabilistic Method*, Springer-Verlag (2002).
6. Christos H. Papadimitriou and Kenneth Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Dover (1998).
7. Robert E. Tarjan, *Data Structures and Algorithms*, SIAM (1983).
8. S. Gill Williamson, *Combinatorics for Computer Science*, Dover (2002).
9. Robin Wilson, *Four Colours Suffice: How the Map Problem was Solved*, Penguin Press (2002).

PART III

Recursion

Recursive thinking plays a fundamental role in combinatorics, theoretical computer science and programming. The next chapter introduces the recursive approach and the following two chapters discuss important applications.

Definition Recursive approach *A* **recursive approach** *to a problem consists of two parts:*

- *The problem is reduced to one or more problems of the same sort which are simpler in some sense.*
- *There is a collection of simplest problems to which all others are reduced after one or more steps. A solution to these simplest problems is given.*

As you might expect, a *recursive algorithm* is one that refers to itself. This seemingly simple notion is extremely important. Recursive algorithms and their translation into recursive procedures and recursive data structures are fundamental in computer science. For example, here's a recursive algorithm for sorting a list. (Sorting a list means putting the items in increasing order.)

- divide the list roughly in half,
- sort each half, and
- "merge" the two sorted halves.

Proof by induction and recursive algorithms are closely related. We'll begin Chapter 7 by examining inductive proofs and recursive equations. Then we'll look briefly at thinking and programming recursively.

Suppose that we have some items that have a "natural" order; e.g., the natural order for student records might be

- alphabetic by name (last name first),
- first by class and, within a class, alphabetic by name, or
- by grade point average with highest first.

We may allow ties. The problem of sorting is as follows: Given a list of items in no particular order, rearrange it so that it is in its natural order. In the event of a tie, the relative order of the tied items is arbitrary. In Chapter 8, we'll study some of the recursive aspects of software and hardware sorting algorithms. Many of these use the "divide and conquer" technique, which often appears in recursive algorithms. We end the chapter with a discussion of this important technique.

One of the most important conceptual tools in computer science is the idea of a rooted plane tree, which we introduced in Section 5.4. This leads naturally to methods for ranking and unranking various classes of unlabeled RP-trees. Many combinatorial algorithms involve "traversing" RP-trees. Grammars can often be thought of in terms of RP-trees, and generating machine code from a higher level language is related to the traversal of such trees. These topics are discussed in Chapter 9.

CHAPTER 7

Induction and Recursion

Introduction

Suppose $\mathcal{A}(n)$ is an assertion that depends on n. We use *induction* to prove that $\mathcal{A}(n)$ is true when we show that

- it's true for the smallest value of n and
- if it's true for everything less than n, then it's true for n.

Closely related to proof by induction is the notion of a recursion. A *recursion* describes how to calculate a value from previously calculated values. For example, $n!$ can be calculated by

$$n! = \begin{cases} 1, & \text{if } n = 0; \\ n \cdot (n-1)!, & \text{otherwise.} \end{cases}$$

We discussed recursions briefly in Section 1.4.

Notice the similarity between the two ideas: There is something to get us started and then each new thing depends on similar previous things. Because of this similarity, recursions often appear in inductively proved theorems as either the theorem itself or a step in the proof. We'll study inductive proofs and recursive equations in the next section.

Inductive proofs and recursive equations are special cases of the general concept of a recursive approach to a problem. Thinking recursively is often fairly easy when one has mastered it. Unfortunately, people are sometimes defeated before reaching this level. We've devoted Section 2 to helping you avoid some of the pitfalls of recursive thinking.

In Section 3 we look at some concepts related to recursive algorithms including proving correctness, recursions for running time, local descriptions and computer implementation.

Not only can recursive methods provide more natural solutions to problems, they can also lead to faster algorithms. This approach, which is often referred to as "divide and conquer," is discussed in Section 4. The best sorting algorithms are of the divide and conquer type, so we'll see a bit more of this in Chapter 8.

7.1 Inductive Proofs and Recursive Equations

The concept of proof by induction is discussed in Appendix A (p. 361). We strongly recommend that you review it at this time. In this section, we'll quickly refresh your memory and give some examples of combinatorial applications of induction. Other examples can be found among the proofs in previous chapters. (See the index under "induction" for a listing of the pages.)

We recall the theorem on induction and some related definitions:

Theorem 7.1 Induction *Let $\mathcal{A}(m)$ be an assertion, the nature of which is dependent on the integer m. Suppose that we have proved $\mathcal{A}(n)$ for $n_0 \le n \le n_1$ and the statement*

"If $n > n_1$ and $\mathcal{A}(k)$ is true for all k such that $n_0 \le k < n$, then $\mathcal{A}(n)$ is true."

Then $\mathcal{A}(m)$ is true for all $m \ge n_0$.

Definition 7.1 *The statement "$\mathcal{A}(k)$ is true for all k such that $n_0 \le k < n$" is called the* **induction assumption** *or* **induction hypothesis** *and proving that this implies $\mathcal{A}(n)$ is called the* **inductive step**. *The cases $n_0 \le n \le n_1$ are called the* **base cases**.

Proof: We now prove the theorem. Suppose that $\mathcal{A}(n)$ is false for some $n \ge n_0$. Let m be the least such n. We cannot have $m \le n_0$ because one of our hypotheses is that $\mathcal{A}(n)$ has been proved for $n_0 \le n \le n_1$. On the other hand, since m is as small as possible, $\mathcal{A}(k)$ is true for $n_0 \le k < m$. By the inductive step, $\mathcal{A}(m)$ is also true, a contradiction. Hence our assumption that $\mathcal{A}(n)$ is false for some n is itself false; in other words, $\mathcal{A}(n)$ is never false. □

Example 7.1 The parity of binary trees The numbers b_n, $n \ge 1$, given by

$$b_1 = 1 \quad \text{and} \quad b_n = b_1 b_{n-1} + b_2 b_{n-2} + \cdots + b_{n-1} b_1 \text{ for } n > 1 \tag{7.1}$$

count the number of "unlabeled full binary RP-trees." We prove this recursion in Example 7.10 and study these trees more in Section 9.3 (p. 259). For now, all that matters is (7.1), not what the b_n count.

Using the definitions, we compute the first few values:

$$b_1 = 1 \quad b_2 = 1 \quad b_3 = 2 \quad b_4 = 5 \quad b_5 = 14 \quad b_6 = 42 \quad b_7 = 132.$$

Most values appear to be even. If you compute b_8, you will discover that it is odd. Since b_1, b_2, b_4 and b_8 are the only odd values with $n \le 8$, we conjecture that b_n is odd if and only if n is a power of 2. Call the conjecture $\mathcal{A}(n)$. How should we choose n_0 and n_1? Since the recursion in (7.1) is only valid for $n > 1$, the case $n = 1$ appears special. Thus we try letting $n_0 = n_1 = 1$ and using the recursion for $n > 1$.

Since $b_1 = 1$ is odd, $\mathcal{A}(1)$ is true. We now consider $n > 1$. If n is odd, let $k = (n-1)/2$ and note that we can write the recursion as

$$b_n \;=\; 2(b_1 b_{n-1} + b_2 b_{n-2} + \cdots + b_k b_{k+1}).$$

Hence b_n is even and no induction was needed. Now suppose n is even and let $k = n/2$. Now our recursion becomes

$$b_n \;=\; 2(b_1 b_{n-1} + b_2 b_{n-2} + \cdots + b_{k-1} b_{k+1}) + b_k^2.$$

Hence b_n is odd if and only if $b_k = b_{n/2}$ is odd. By the induction assumption, $b_{n/2}$ is odd if and only if $n/2$ is a power of 2. Since $n/2$ is a power of 2 if and only if n is a power of 2, we are done. □

Example 7.2 The Fibonacci numbers One definition of the Fibonacci numbers is

$$F_0 = 0, \quad F_1 = 1, \quad \text{and} \quad F_{n+1} = F_n + F_{n-1} \quad \text{for } n > 0. \tag{7.2}$$

We want to prove that

$$F_n \;=\; \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \quad \text{for } n \geq 0. \tag{7.3}$$

Let that be $\mathcal{A}(n)$. Since (7.2) is our only information, we'll use it to prove (7.3). We must either think of our induction in terms of proving $\mathcal{A}(n+1)$ or rewrite the recursion as $F_n = F_{n-1} + F_{n-2}$. We'll use the latter approach Since the recursion starts at $n+1 = 2$, we'll have to prove $\mathcal{A}(0)$ and $\mathcal{A}(1)$ separately. Hence $n_0 = 0$ and $n_1 = 1$ in Theorem 7.1. Since

$$\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^0 - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^0 \;=\; 1 - 1 \;=\; 0,$$

$\mathcal{A}(0)$ is true. Since

$$\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^1 - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^1 \;=\; 1,$$

$\mathcal{A}(1)$ is true.

Now for the induction. We want to prove (7.3) for $n \geq 1$. By the recursion, $F_n \;=\; F_{n-1} + F_{n-2}$. Now use $\mathcal{A}(n-1)$ and $\mathcal{A}(n-2)$ to replace F_{n-1} and F_{n-2}. Thus

$$F_n \;=\; \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n-1} + \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n-2},$$

and so we want to prove that

$$\begin{aligned}\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \;=\; & \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n-1} \\ & + \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n-2}.\end{aligned}$$

Consider the three terms that involve $1+\sqrt{5}$. Divide by $\left(\frac{1-\sqrt{5}}{2}\right)^{n-2}$ and multiply by $\sqrt{5}$ to see that they combine correctly if

$$\left(\frac{1+\sqrt{5}}{2}\right)^2 \;=\; \frac{1+\sqrt{5}}{2} + 1,$$

which is true by simple algebra. The three terms with $1-\sqrt{5}$ are handled similarly. □

Example 7.3 Disjunctive form for Boolean functions We will consider functions with domain $\{0,1\}^n$ and range $\{0,1\}$. A typical function is written $f(x_1,\ldots,x_n)$. These functions are called *Boolean functions* on n variables. With 0 interpreted as "false" and 1 as "true," we can think of $x_1,\ldots,x_n$ as statements which are either true or false. In this case, f can be thought of as a complicated statement built from $x_1,\ldots,x_n$ which is true or false depending on the truth and falsity of the x_i's.

For $y_1,\ldots,y_k \in \{0,1\}$, $y_1y_2\cdots y_k$ is normal multiplication; that is,

$$y_1y_2\cdots y_k \;=\; \begin{cases} 1, & \text{if } y_1=y_2=\cdots=y_k=1;\\ 0, & \text{otherwise.}\end{cases}$$

Define

$$y_1+y_2+\cdots+y_m \;=\; \begin{cases} 0, & \text{if } y_1=y_2=\cdots=y_k=0;\\ 1, & \text{otherwise.}\end{cases}$$

With the true-false interpretation, multiplication corresponds to "and" and + corresponds to "or." Define $x'=1-x$, the *complement* of x.

A function f is said to be written in *disjunctive form* if

$$f(x_1,\ldots,x_n) \;=\; A_1+\cdots+A_k, \tag{7.4}$$

where each A_j is the product of terms, each of which is either an x_i or an x_i'. For example, let $g(x_1,x_2,x_3)$ be 1 if exactly two of x_1, x_2 and x_3 are 1, and 0 otherwise. Then

$$g(x_1,x_2,x_3) \;=\; x_1x_2x_3' + x_1x_2'x_3 + x_1'x_2x_3$$

and

$$g(x_1,x_2,x_3)' \;=\; x_1'x_2' + x_1'x_3' + x_2'x_3' + x_1x_2x_3$$

If $k=0$ in (7.4) (i.e., no terms present), then it is interpreted to be 0 for all $x_1,\ldots,x_n$.

We will prove

Theorem 7.2 *Every Boolean function can be written in disjunctive form.*

Let $\mathcal{A}(n)$ be the theorem for Boolean functions on n variables. There are $2^2=4$ Boolean functions on 1 variable. Here are the functions and disjunctive forms for them:

$(f(0),f(1))$	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
form	x_1x_1'	x_1	x_1'	x_1+x_1'

This proves $\mathcal{A}(1)$.

For $n>1$ we have

$$f(x_1,\ldots,x_n) \;=\; \big(g_0(x_1,\ldots,x_{n-1})\,x_n'\big) + \big(g_1(x_1,\ldots,x_{n-1})\,x_n\big), \tag{7.5}$$

where $g_k(x_1,\ldots,x_{n-1}) = f(x_1,\ldots,x_{n-1},k)$. To see this, note that when $x_n=0$ the right side of (7.5) is $(g_0\cdot 1)+(g_1\cdot 0)=g_0=f$ and when $x_n=1$ it is $(g_0\cdot 0)+(g_0\cdot 1)=g_1=f$.

By the induction assumption, both g_0 and g_1 can be written in disjunctive form, say

$$g_0 \;=\; A_1+\cdots+A_a \qquad \text{and} \qquad g_1 \;=\; B_1+\cdots+B_b. \tag{7.6}$$

We claim that

$$(C_1+\cdots+C_c)y \;=\; C_1y+\cdots+C_cy. \tag{7.7}$$

If this is true, then it can be used in connection with (7.6) in (7.5) to complete the inductive step.

To prove (7.7), notice that

$$\text{(the left side of (7.7) equals 1)} \quad \text{if and only if} \quad (y=1 \text{ and some } C_i=1).$$

This is equivalent to

$$\text{(the left side of (7.7) equals 1)} \quad \text{if and only if} \quad \text{(some } C_i y = 1\text{)}.$$

however,

$$\text{(the right side of (7.7) equals 1)} \quad \text{if and only if} \quad \text{(some } C_i y = 1\text{)}.$$

This proves that the left side of (7.7) equals 1 if and only if the right side equals 1. Thus (7.7) is true. □

Suppose you have a result that you are trying to prove. If you are unable to do so, you might try to prove a bit less because proving less should be easier. That is not always true for proofs by induction. Sometimes it is easier to prove more! How can this be? The statement $\mathcal{A}(n)$ is not just the thing you want to prove, it is also the assumption that you have to help you prove $\mathcal{A}(m)$ for $m > n$. Thus, a stronger inductive hypothesis gives you more to prove *and* more to prove it with. This should be emphasized:

> **Principle More may be better** *If the induction hypothesis seems to weak to carry out an inductive proof, consider trying to prove a stronger theorem.*

We've already encountered this in proving Theorem 6.2 (p. 153). We had wanted to prove that every connected graph has a lineal spanning tree. We might have used

> $\mathcal{A}_1(n)$: "If G is an n-vertex connected graph, it has a lineal spanning tree."

Instead we used the stronger statement

> $\mathcal{A}_2(n)$: "If G is an n-vertex connected graph containing the vertex r, it has a lineal spanning tree with root r."

If you try to prove $\mathcal{A}_1(n)$ by induction, you'll soon run into problems. Try it. The following example illustrates the usefulness of generalizing the hypothesis for some inductive proofs.

Example 7.4 Graphs and Ramsey Theory Let k be a positive integer and let $G = (V, E)$ be an arbitrary simple graph. Can we find a subset $S \subseteq V$ such that $|S| = k$ and either

- for all $x, y \in S$, we have $\{x, y\} \in E$ or
- for all $x, y \in S$, we have $\{x, y\} \notin E$?

If $|V|$ is too small, e.g., $|V| < k$, the answer is obviously "No." Instead, we might ask, "Is there an $N(k)$ such that there exists an S with the above properties whenever $|V| \geq N(k)$?" You should be able to see that, if we find some value which works for $N(k)$, then any larger value will also work.

It's easy to see that we can choose $N(2) = 2$: Pick any two $x, y \in V$, let $S = \{x, y\}$. Since $\{x, y\}$ is either an edge in G or is not, we are done.

Let's try to show that $N(3)$ exists and find the smallest possible value we can choose for it.

You should find a simple graph G with $|V| = 5$ for which the result is false when $k = 3$; that is, for any set of three vertices in G there is at least one pair that are joined by an edge and at least one pair that are not joined by an edge. Having done this, you've shown that, if $N(3)$ exists it must be greater than 5.

We now prove that we may take $N(3) = 6$. Select any $v \in V$. Of the remaining five or more vertices in V there must be at least three that are joined to v or at least three that are not joined to v. We do the first case and leave the latter for you. Let x_1, x_2 and x_3 be three vertices joined to v. If $\{x_i, x_j\} \in E$, then all pairs of vertices in $\{v, x_i, x_j\}$ are joined by edges and we are done. If $\{x_i, x_j\} \notin E$ for all i and j, then none of the pairs of vertices in $\{x_1, x_2, x_3\}$ are joined by edges and, again, we are done. We have shown that we may take $N(3) = 6$.

Since the proof that $N(3)$ exists involved reduction to a smaller situation, it suggests that we might be able to prove the existence of $N(k)$ by induction on k. How would this work? Here's a brief sketch. We'd select $v \in V$ and note the existence of a large enough set all of whose vertices were

either joined to v by edges or not joined to v. As above, we could assume the former case. We now want to know that there exist either $k-1$ vertices all joined to v or k vertices not joined to v. This requires the induction assumption, but we are stuck because we are looking for either a set of size $k-1$ or one of size k, which are two different sizes. We can get around this problem by strengthening the statement of the theorem to allow two different sizes. Here's the theorem we'll prove.

Theorem 7.3 A special case of Ramsey's Theorem *There exists a function $N(k_1, k_2)$, defined for all positive integers k_1 and k_2, such that, for all simple graphs $G = (V, E)$ with at least $N(k_1, k_2)$ vertices, there is a set $S \subseteq V$ such that either*

- *$|S| = k_1$ and $\{x, y\} \in E$ for all $x \neq y$ both in S, or*
- *$|S| = k_2$ and $\{x, y\} \notin E$ for all $x \neq y$ both in S.*

In fact, we may define an acceptable $N(k_1, k_2)$ recursively by

$$N(k_1, k_2) = \begin{cases} 1, & \text{if } k_1 = 1; \\ 1, & \text{if } k_2 = 1; \\ N(k_1 - 1, k_2) + N(k_1, k_2 - 1) + 1, & \text{otherwise.} \end{cases} \tag{7.8}$$

If you have mistakenly assumed that $N(k_1, k_2)$ is uniquely defined—an easy error to make—the phrase "an acceptable $N(k_1, k_2)$" in the theorem probably bothers you. Look back over our earlier discussion of $N(k)$, which is the case $k_1 = k_2$. We said that $N(k)$ was any number such that if we had at least $N(k)$ vertices something was true, and we observed that if some value worked for $N(k)$, any larger value would also work. Of course we could look for the smallest possible choice for $N(k)$. We found that this is 2 when $k = 2$ and is 6 when $k = 3$. The theorem does not claim that the recursion (7.8) gives the smallest possible choice for $N(k_1, k_2)$. In fact, it tends to give numbers that are much too big. Since we showed earlier that the smallest possible value for $N(3, 3)$ is 6, you might mistakenly think that finding the smallest is easy. In fact, the smallest possible value of $N(k_1, k_2)$ is unknown for almost all (k_1, k_2).

Proof: We'll use induction on $n = k_1 + k_2$.

Before starting the induction step, we'll do the case in which $k_1 = 1$ or $k_2 = 1$ (or both). If $k_1 = 1$, choose $s \in V$ and set $S = \{s\}$. The theorem is trivially true because there are no $x \neq y$ in S. Similarly, it is trivially true if $k_2 = 1$.

We now carry out the inductive step. By the previous paragraph, we can assume that $k_1 > 1$ and $k_2 > 1$. Choose $v \in V$ and define

$$V_1 = \Big\{ x \in V - \{v\} \;\Big|\; \{x, v\} \in E \Big\}$$
$$V_2 = \Big\{ x \in V - \{v\} \;\Big|\; \{x, v\} \notin E \Big\}.$$

It follows by (7.8) that either $|V_1| \geq N(k_1 - 1, k_2)$ or $|V_2| \geq N(k_1, k_2 - 1)$. We assume the former. (The argument for the latter case would be very similar to the one we are about to give.)

Look at the graph $(V_1, E \cap \mathcal{P}_2(V_1))$. Since $|V_1| \geq N(k_1 - 1, k_2)$, it follows from the inductive hypothesis that there is a set $S' \subseteq V_1$ such that either

- $|S'| = k_1 - 1$ and $\{x, y\} \in E$ for all $x \neq y$ both in S', or
- $|S'| = k_2$ and $\{x, y\} \notin E$ for all $x \neq y$ both in S'.

If the former is true, let $S = S' \cup \{v\}$; otherwise, let $S = S'$. This completes the proof.

A more general form of Ramsey's Theorem asserts that there exists a function $N_r(k_1, \ldots, k_d)$ such that for all V with $|V| \geq N_r(k_1, \ldots, k_d)$ and all f from $\mathcal{P}_r(V)$ to $\underline{d}$, there exists an $i \in \underline{d}$ and a set $S \subseteq V$ such that $|S| = k_i$ and $f(e) = i$ for all $e \in \mathcal{P}_r(S)$. The theorem we proved is the special case $N_2(k_1, k_2)$ and $f(e)$ is 1 or 2 according as $e \in E$ or $e \notin E$. Although the more general statement looks fairly complicated, its no harder to prove than the special case—provided you don't get lost in all the notation. You might like to try proving it. □

Exercises

In these exercises, indicate clearly

(i) what $\mathcal{A}(n)$ is,

(ii) what the inductive step is and

(iii) where the inductive hypothesis is used.

7.1.1. Indicate (i)–(iii) in the proof of the rank formula (Theorem 3.1 (p. 76)).

7.1.2. Indicate (i)–(iii) in the proof of the greedy algorithm for unranking in Section 3.2 (p. 80).

7.1.3. Do Exercise 1.3.11.

7.1.4. For $n \geq 0$, let D_n be the number of derangements (permutations with no fixed points) of an n-set. By convention, $D_0 = 1$. The next few values are $D_1 = 0$, $D_2 = 1$, $D_3 = 2$ and $D_4 = 9$. Here are some statements about D_n.

(i) $D_n = nD_{n-1} + (-1)^n$ for $n \geq 1$.

(ii) $D_n = (n-1)(D_{n-1} + D_{n-2})$ for $n \geq 2$.

(iii) $D_n = n! \sum_{k=0}^{n} \frac{(-1)^k}{k!}$.

(a) Use (i) to prove (ii). (Induction is not needed.)

(b) Use (ii) to prove (i).

(c) Use (i) to prove (iii).

(d) Use (iii) to prove (i). (Induction is not needed)

7.1.5. Write the following Boolean functions in disjunctive form. The functions are given in two-line form.

(a) $\begin{pmatrix} 0,0 & 0,1 & 1,0 & 1,1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$.

(b) $\begin{pmatrix} 0,0 & 0,1 & 1,0 & 1,1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$.

(c) $\begin{pmatrix} 0,0,0 & 0,0,1 & 0,1,0 & 0,1,1 & 1,0,0 & 1,0,1 & 1,1,0 & 1,1,1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$.

(d) $\begin{pmatrix} 0,0,0 & 0,0,1 & 0,1,0 & 0,1,1 & 1,0,0 & 1,0,1 & 1,1,0 & 1,1,1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$.

7.1.6. Write the following Boolean functions in disjunctive form.

(a) $(x_1 + x_3)(x_2 + x_4)$.

(b) $(x_1 + x_2)(x_1 + x_3)(x_2 + x_3)$.

7.1.7. A Boolean function f is written in *conjunctive form* if $f = A_1 A_2 \cdots$, where A_i is the "or" of terms each of which is either an x_i or an x_i'. Prove that every Boolean function can be written in conjunctive form.
Hint. The proof parallels that in Example 7.3. The hardest part is probably finding the equation that replaces (7.5).

7.1.8. Part II contains a variety of results that are proved by induction. Some appear in the text and some in the exercises. Write careful inductive proofs for each of the following.

(a) Every connected graph has a lineal spanning tree.

(b) The number of ways to color a simple graph G with x colors is a polynomial in x. (Do this by deletion and contraction.)

(c) Euler's relation: $v - e + f = 2$.

(d) Every planar graph can be colored with 5 colors.

(e) Using the fact that every tree has a leaf, prove that an n-vertex tree has exactly $n - 1$ edges.

(f) Every n-vertex connected graph has at least $n - 1$ edges.

*7.1.9. Using the definition of the Fibonacci numbers in Example 7.2, prove that

$$F_{n+k+1} \;=\; F_{n+1}F_{k+1} + F_nF_k \quad \text{for} \quad k \geq 0 \text{ and } n \geq 0.$$

Do *not* use formula (7.3).
Hint. You may find it useful to note that $n + k + 1 \;=\; (n - 1) + (k + 1) + 1$.

7.2 Thinking Recursively

A *recursive formula* tells us how to compute the value we are interested in terms of earlier ones. (The "earliest" values are specified separately.) How many can you recall from previous chapters? A *recursive definition* describes more complicated instances of a concept in terms of simpler ones. (The "simplest" instances are specified separately.) These are examples of the recursive approach, which we defined at the beginning of this part:

Definition 7.2 Recursive approach *A* **recursive approach** *to a problem consists of two parts:*

1. *The problem is reduced to one or more problems of the same kind which are simpler in some sense.*
2. *There is a set of simplest problems to which all others are reduced after one or more steps. Solutions to these simplest problems are given.*

This definition focuses on tearing down (reduction to simpler cases). Sometimes it may be easier or better to think in terms of building up (construction of bigger cases). We can simply turn Definition 7.2 on its head:

Definition 7.3 Recursive solution *We have a* **recursive solution** *to the problem (proof, algorithm, data structure, etc.) if the following two conditions hold.*

1. *The set of simplest problems can be dealt with (proved, calculated, sorted, etc.).*
2. *The solution to any other problem can be built from solutions to simpler problems, and this process eventually leads back to the simplest problems.*

Let's look briefly at some examples where recursion can be used. Suppose that we are given a collection of things and a problem associated with them. Examples of things and problems are

- assertions $\mathcal{A}(n)$ that we want to prove;
- the binomial coefficients $C(n, k)$ that we want to compute using $C(n, k) \;=\; C(n - 1, k) + C(n - 1, k - 1)$ from Section 1.4;
- the recursion $D_n \;=\; (n - 1)(D_{n-1} + D_{n-2})$ for derangements that we want to prove by a direct combinatorial argument;
- lists to sort;
- RP-trees that we want to define.

Suppose we have some binary relation between these things which we'll denote by "simpler than." If there is nothing simpler than a thing X, we call X "simplest." There may be several simplest things. In the examples just given, we'll soon see that the following notions are appropriate.

- $\mathcal{A}(n)$ is simpler than $\mathcal{A}(m)$ if $n < m$ and $\mathcal{A}(1)$ is the simplest thing.
- $C(n, k)$ is simpler than $C(m, j)$ if $n < m$ and the $C(0, k)$'s are the simplest things.

- D_n is simpler that D_m if $n > m$ and the simplest things are D_0 and D_1.
- One list is simpler than another if it contains fewer items and the lists with one item are the simplest things.
- One tree is simpler than another if it contains less vertices and the one vertex tree is the simplest.

Example 7.5 The induction theorem The induction theorem in the previous section solves the problem of proving $\mathcal{A}(n)$ recursively. There is only one simplest problem: $\mathcal{A}(1)$. We are usually taking a reduction viewpoint when we prove something by induction. □

Example 7.6 Calculating binomial coefficients Find a method for calculating the binomial coefficients $C(n,k)$. As indicated above, we let the simplest values be those with $n = 0$. From Chapter 1 we have

- $C(0,k) = \begin{cases} 1, & \text{if } k = 0; \\ 0, & \text{otherwise.} \end{cases}$
- $C(n,k) = C(n-1,k-1) + C(n-1,k).$

This solves the problem recursively. □

Is the derivation of the binomial coefficient recursion done by reduction or construction? We can derive it by dividing the k-subsets of $\underline{n}$ into those that contain n and those that do not. This can be regarded as reduction or construction. Such ambiguities are common because the two concepts are simply different facets of the same thing. Nevertheless, it is useful to explore reduction versus construction in problems so as to gain facility with solving problems recursively. We do this now for derangements.

Example 7.7 A recursion for derangements A derangement is a permutation without fixed points and D_n is the number of derangements of an n-set. In Exercise 7.1.4 the recursion

$$D_n = (n-1)(D_{n-1} + D_{n-2}) \quad \text{for} \quad n \geq 2 \tag{7.9}$$

with initial conditions $D_0 = 1$ and $D_1 = 0$ was stated without proof. We now give a derivation using reduction and construction arguments.

Look at a derangement of $\underline{n}$ in cycle form. Since a derangement has no fixed points, no cycles have length 1. We look at the cycle of the derangement that contains n.

(a) If this cycle has length 2, throw out the cycle.

(b) If this cycle has length greater than 2, remove n from the cycle.

In case (a), suppose k is in the cycle with n. We obtain every derangement of $\underline{n} - \{k,n\}$ exactly once. Since there are $n-1$ possibilities for k, (a) contributes $(n-1)D_{n-2}$ to the count. This is a reduction point of view.

In case (b), we obtain derangements of $\underline{n-1}$. To find the contribution of (b), it may be easier to take a construction view: Given a derangement of $\underline{n-1}$ in cycle form, we choose which of the $n-1$ elements to insert n after. This gives a contribution of $(n-1)D_{n-1}$

The initial conditions and the range of n for which (7.9) is valid can be found by examining our argument. There are two approaches:

- We could take the view that derangements only make sense for $n \geq 1$ and so (7.9) is used when $n > 3$, with initial conditions $D_1 = 0$ and $D_2 = 1$.
- We could look at the argument used to derive the recursion and ask how we should define D_n for $n < 1$ so that the argument makes sense. Note that for $n = 1$, the values of D_0 and D_{-1} don't matter since the recursion gives

  $$D_1 = (1-1)(D_0 + D_{-1}) = 0(D_0 + D_{-1}) = 0,$$

  which is correct. What about $n = 2$? We look at (a) and (b) separately.

(a) We want to get the derangement $(1, 2)$, so we need $D_0 = 1$.

(b) This should give zero since there is no derangement of $\underline{2}$ containing a cycle of length exceeding 2. Thus we need $D_1 = 0$, which we have.

To summarize, we can use (7.9) for $n \geq 1$ with the initial conditions $D_0 = 1$. □

Example 7.8 Merge sorting Merge sorting can be described as follows.

1. The lists containing just one item are the simplest and they are already sorted.
2. Given a list of $n > 1$ items, choose k with $1 \leq k < n$, sort the first k items, sort the last $n - k$ items and merge the two sorted lists.

This algorithm builds up a way to sort an n-list out of procedures for sorting shorter lists. Note that we have not specified how the first k or last $n - k$ items are to be sorted, we simply assume that it has been done. Of course, an obvious way to do this is to simply apply our merge sorting algorithm to each of these sublists.

Let's implement the algorithm using people rather than a computer. Imagine training a large number of obedient people to carry out two tasks: splitting a list for other people to sort and merging two lists. We give one person the unsorted list and tell him to sort it using the algorithm and return the result to us.

What happens? Anyone who has a list with only one item returns it unchanged to the person he received it from. This is Case 1 in Definition 7.3 (p. 204) (and also in the algorithm). Anyone with a list having more than one item splits it and gives each piece to a person who has not received a list, telling each person to sort it and return the result. When the results have been returned, this person merges the two lists and returns the result to whoever gave him the list. If there are enough obedient people around, we'll eventually get our answer back.

Notice that no one needs to pay any attention to what anyone else is doing to a list. □

We now look at one of the most important recursive definitions in computer science.

Example 7.9 Defining rooted plane trees recursively Rooted plane trees (RP-trees) were defined in Section 5.4 (p. 136). Here is a recursive constructive definition of RP-trees.

- A single vertex, which we call the root, is an RP-tree.
- If $T_1, \ldots, T_k$ is an ordered list of RP-trees with roots $r_1, \ldots, r_k$ and no vertices in common, then an RP-tree can be constructed by choosing an unused vertex r to be the root, letting its ith child be r_i and forgetting that $r_1, \ldots, r_k$ were called roots.

This is a more compact definition than the nonconstructive one given in Section 5.4. This approach to RP-trees is very important for computer science. We'll come back to it in the next section.

We should, and will, prove that this definition is equivalent to that in Section 5.4. In other words, our new "definition" should not be regarded as a definition but, rather, as a theorem—you can only define something once!

Define an *edge* to be any set of two vertices in which one vertex is the child of the other. Note that the recursive definition insures that the graph is connected and the use of distinct vertices eliminates the possibility of cycles. Thus, the "definition" given here leads to a rooted, connected, simple graph without loops. Furthermore, the edges leading to a vertex's sons are ordered. Thus we have an RP-tree. To actually prove this carefully, one must use induction on the number of vertices. This is left as an exercise.

It remains to show that every RP-tree, as defined in Section 5.4, can be built by the method described in the recursive "definition" given above. One can use induction on the number of vertices. It is obvious for one vertex. Remove the root vertex and note that each child of the root now becomes the root of an RP-tree. By the induction hypothesis, each of these can be built by our recursive

process. The recursive process allows us to add a new root whose children are the roots of these trees, and this reconstructs the original RP-tree.

Here is another definition of an RP-tree.

- A single vertex, which we call the root, is an RP-tree.
- If T_1 and T_2 are RP-trees with roots r_1 and r_2 and no vertices in common, then an RP-tree can be constructed by connecting r_1 to r_2 with an edge, making r_2 the root of the new tree and making r_1 the leftmost child of r_2.

We leave it to you to prove that this is equivalent to the previous definition □

Example 7.10 Recursions for rooted plane trees Rooted trees in which each non-leaf vertex has exactly two edges leading away from the root is called a full binary tree. By replacing k in the previous example with 2, we have a recursive definition of them: A full binary RP-tree is either a single vertex or a new root vertex joined to two full binary RP-trees.

As noted in Section 1.4 (p. 32), recursive constructions lead to recursions. Let's use the previous recursive definition to get a recursion for full binary trees. Suppose we require that any node that is not a leaf have exactly two children. Let b_n be the number of such trees that have n leaves. From the recursive definition, we have $b_1 = 1$ for the single vertex tree. Since the recursive construction gives us each tree *exactly once*, we have

$$b_n \;=\; b_1 b_{n-1} + b_2 b_{n-2} + \cdots + b_{n-1} b_1 \;=\; \sum_{j=1}^{n-1} b_j b_{n-j} \qquad \text{for } n > 1.$$

To see why this is so, apply the Rules of Sum and Product: First, partition the problem according to the number of leaves in T_1, which is the j in our formula. Second, for each case choose T_1 and then choose T_2, which gives us the term $b_j b_{n-j}$.

If we try the same approach for general rooted plane trees, we have two problems. First, we had better not count by leaves since there are an infinite number of trees with just one leaf, namely trees with of the form •—•—•—⋯—•. Second, the fact the the definition involves $T_1, \ldots, T_k$ where k can be any positive integer makes the recursion messy: we'd have to sum over all such k and for each k we'd have a product $t_{j_1} \cdots t_{j_k}$ to sum over all j's such that $j_1 + \cdots + j_k$ has the appropriate value.

The first problem is easy to fix: let t_n be the number of rooted plane trees with n vertices. The second problem requires a new recursive construction, which means we have to be clever. We use the construction in the last paragraph of the previous example. We then have $t_1 = 1$ and, for $n > 1$, $t_n = \sum_{j=1}^{n-1} t_j t_{n-j}$, because if the constructed tree has n vertices and T_1 has j vertices, then T_2 has $n - j$ vertices. Notice that t_n and b_n satisfy the same recursion with the same initial conditions. Since the recursion lets us compute all values recursively, it follows that $t_n = b_n$. (Alternatively, you could prove $b_n = t_n$ using the recursion and induction on n.) Actually, there is a slight gap here: we didn't prove that the new recursive definition gives all rooted plane trees and gives them exactly once. We leave it to you to convince yourself of this. □

A *recursive algorithm* is an algorithm that refers to itself when it is executing. As with any recursive situation, when an algorithm refers to itself, it must be with "simpler" parameters so that it eventually reaches one of the "simplest" cases, which is then done without recursion. Our recursion for $C(n, k)$ can be viewed as a recursive algorithm. Our description of merge sorting in Examples 7.8 gives a recursive algorithm if the sorting required in Step 2 is done by using the algorithm itself. Let's look at one more example illustrating the recursive algorithm idea.

Example 7.11 A recursive algorithm Suppose you are interested in listing all sequences of length eight, consisting of four zeroes and four ones. Suppose that you have a friend who does this sort of thing, but will only make such lists if the length of the sequence is seven or less. "Nope," he says, "I can't do it—the sequence is too long." There is a way to trick your friend into doing it. First give him the problem of listing all sequences of length seven with three ones. He doesn't mind, and gives you the list 1110000, 1011000, 0101100, etc. that he has made. You thank him politely, sneak off, and put a "1" in front of every sequence in the list he has given you to obtain 11110000, 11011000, 10101100, etc. Now, you return to him with the problem of listing all strings of length seven with four ones. He returns with the list 1111000, 0110110, 0011101, etc. Now you thank him and sneak off and put a "0" in front of every sequence in the list he has given you to obtain 01111000, 00110110, 00011101, etc. Putting these two lists together, you have obtained the list you originally wanted.

How did your friend produce these lists that he gave you? Perhaps he had a friend that would only do lists of length 6 or less, and he tricked this friend in the same way you tricked him! Perhaps the "6 or less" friend had a "5 or less friend" that he tricked, etc. If you are sure that your friend gave you a correct list, it doesn't really matter how he got it. □

These examples are rather easy to follow, but what happens if we look into them more deeply? We might ask just how $C(15, 7)$ is calculated in terms of the simplest values $C(0, k)$ without specifying any of the intermediate values. We might ask just what all of our trained sorters are doing. We might ask how your friend got his list of sequences.

This kind of analysis is often tempting to do when we are debugging recursive algorithms. It is almost always the wrong thing to do. Asking about such details usually leads to confusion and gets one so off the track that it is even harder to convince oneself that the algorithm is correct.

Why is it unnecessary to "unwind" the recursion in this fashion? If Case 2 of our recursive solution as given by Definition 7.3 (p. 204) correctly describes what to do, *assuming that the simpler problems have been done correctly*, then our recursive solution works! This can be demonstrated the way induction was proved: If the solution fails, there must be a problem for which it fails such that it succeeds for all simpler problems. If this problem is simplest, it contradicts Case 1 in Definition 7.3. If this problem is not simplest, it contradicts Case 2 in Definition 7.3 since all simpler problems have been dealt with. Thus our assumption that the solution fails has led to a contradiction. It is important to understand this proof since it is the theoretical basis for recursive methods. To summarize:

Principle Thinking recursively *Carefully verify the two parts of Definition 7.2 or of Definition 7.3. Avoid studying the results of iterating the recursive solution.*

If you are just learning about recursion, you may find it difficult to believe that this general strategy will work without seeing particular solutions where the reduction to the simplest cases is laid out in full detail. In even a simple recursive solution, it is likely that you'll become confused by the details, even if you're accustomed to thinking recursively. If you agree that the proof in the previous paragraph is correct, then such detail is not needed to see that the algorithm is correct. It is *very important* to realize this and to avoid working through the stages of the recursive solution back to the simplest things.

If for some reason you must work backwards through the recursive stages, do it gingerly and carefully. When must you work backwards like this?

- For some reason you may be skipping over an error in your algorithm and so are unable to correct it. The unwinding process can help, probably not because it will help you find the error directly but because it will force you to examine the algorithm more closely.
- You may wish to replace the recursive algorithm with a nonrecursive one. That may require a much deeper understanding of what happens as the recursion is iterated.

The approach of focusing on the two steps of an inductive solution is usually difficult for beginners to maintain. Resist the temptation to abandon it! This does not mean that you should avoid details, but the details you should concern yourself with are different:

- "Is every solution built from simplest solutions, and have I handled the simplest solutions properly?" If not, then the foundation of your recursively built edifice is rotten and the entire structure will collapse.
- "Is my description of how to use simpler solutions to build up more complicated ones correct?"
- "If this is an algorithm, have I specified all the recursive parameters?"

This last point will be dealt with in the next section where we'll discuss implementation.

This does not mean one should never look at the details of a recursion. There are at least two situations in which one does so. First, one may wish to develop a nonrecursive algorithm. Understanding the details of how the recursive algorithm works may be useful. Second, one may need to reduce the amount of storage a recursive algorithm requires.

Exercises

7.2.1. We will prove that all positive integers are equal. Let $\mathcal{A}(n)$ be the statement "All positive integers that do not exceed n are equal." In other words, "If p and q are integers between 1 and n inclusive, then $p = q$." Since 1 is the only positive integer not exceeding 1, $\mathcal{A}(1)$ is true. For $n > 1$, we now assume $\mathcal{A}(n-1)$ and prove $\mathcal{A}(n)$. If p and q are positive integers not exceeding n, let $p' = p - 1$ and $q' = q - 1$. Since p' and q' do not exceed $n - 1$, we have $p' = q'$ by $\mathcal{A}(n-1)$. Thus $p = q$. This proves $\mathcal{A}(n)$. Where is the error?

7.2.2. What is wrong with the following proof that every graph can be drawn in the plane in such a way that no edges intersect? Let $\mathcal{A}(n)$ be the statement for all graphs with n vertices. Clearly $\mathcal{A}(1)$ is true. Let G be a graph with vertices $v_1, \ldots, v_n$. Let G_1 be the subgraph induced by $v_2, \ldots, v_n$ and let G_n be the subgraph induced by $v_1, \ldots, v_{n-1}$. By the induction assumption, we can draw both G_1 and G_n in the plane. After drawing G_n, add the vertex v_n near v_1 and use the drawing of G_1 to see how to connect v_n to the other vertices.

7.2.3. What is wrong with the following proof that all positive integers are interesting? Suppose the claim is false and let n be the smallest positive integer which is not interesting. That is an interesting fact about n, so n is interesting!

7.2.4. What is wrong with the following method for doing this exercise? Ask someone else in the class who will tell you the answer if he/she knows it. If that person knows it, you are done; otherwise that person can use this method to find the answer and so you are done anyway.
Remark: Of course it could be wrong morally because it may be cheating. For this exercise, you should find another reason.

7.2.5. This relates to Example 7.9. Fill in the details of the proof of the equivalence of the two definitions of RP-trees.

7.3 Recursive Algorithms

We'll begin this section by using merge sort to illustrate how to obtain information about a recursive algorithm. In this case we'll look at proof of correctness and a recursion for running time. Next we'll turn our attention to the local description of a recursive procedure. What are the advantages of thinking locally?

- Simplicity: By thinking locally, we can avoid the quagmire that often arises in attempting to unravel the details of the recursion. To avoid the quagmire: *Think locally*, but remember to deal with initial conditions.
- Implementation: A local description lays out in graphical form a plan for coding up a recursive algorithm.
- Counting: One can easily develop a recursion for counting structures, operations, etc.
- Proofs: A local description lays out the plan for an inductive proof.

Finally, we'll turn our attention to the problem of how recursive algorithms are actually implemented on a computer. If you are not programming recursive algorithms at present, you may think of the implementation discussion as an extended programming note and file it away for future reference after skimming it.

Obtaining Information: Merge Sorting

Here's an algorithm for "merge sorting" the sequence s and storing the answer in the sequence t.

```
Procedure SORT(s_1, ..., s_n into t_1, ..., t_n)
     If (n = 1)
          t_1 = s_1
          Return
     End if
     Let m be n/2 with remainder discarded
     SORT(s_1, ..., s_m into u_1, ..., u_m)
     SORT(s_{m+1}, ..., s_n into v_1, ..., v_{n-m})
     MERGE(sequences u and v into t)
     Return
End
```

How do we know it doesn't run forever—an "infinite loop"? How do we know it's correct? How long does it take to run? As we'll see, we can answer such questions by making modifications to the algorithm.

The infinite loop question can be dealt with by verifying the conditions of Definition 7.2 (p. 204). For the present algorithm, the complexity of the problem is the length of the sequence and the simplest case is a 1-long sequence. The algorithm deals directly with the simplest case. Other cases are reduced to simpler ones because a list is divided into shorter lists.

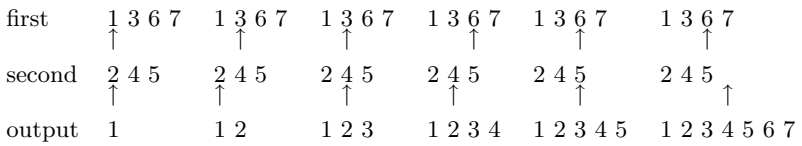


Figure 7.1 Merging two sorted lists. The first and second lists are shown with their pointers at each step.

Example 7.12 The merge sort algorithm is correct One way to prove program correctness is to insert claims into the code and then prove that the claims are correct. For recursive algorithms, this requires induction. We'll assume that the `MERGE` algorithm is known to be correct. (Proving that is an another problem.) Here's our code with comments added for proving correctness.

```
Procedure SORT(s_1, ..., s_n into t_1, ..., t_n)
     If (n = 1)
          t_1 = s_1
          Return                                   /* t is sorted */
     End if
     Let m be n/2 with remainder discarded
     SORT(s_1, ..., s_m into u_1, ..., u_m)        /* u is sorted */
     SORT(s_{m+1}, ..., s_n into v_1, ..., v_{n-m}) /* v is sorted */
     MERGE(sequences u and v into t
     Return                                        /* t is sorted */
End
```

We now use induction on n to prove

$$\mathcal{A}(n) \quad = \quad \text{"When a comment is reached in sorting an } n\text{-list, it is true."}$$

For $n = 1$, only the first comment is reached and it is clearly true since there is only one item in the list. For $n > 1$, the claims about u and v are true by $\mathcal{A}(m)$ and $\mathcal{A}(n-m)$. Also, the claim about t is true by the assumption that `MERGE` runs correctly. □

Example 7.13 The running time for a merge sort How long does the merge sort algorithm take to run? Let's ignore the overhead in computing m, subroutine calling and so forth and focus on the part that takes the most time: merging.

Suppose that $u_1 \leq \ldots \leq u_i$ and $v_1 \leq \ldots \leq v_j$ are two sorted lists. We can merge these two ordered lists into one ordered list very simply by moving pointers along the two lists and comparing the elements being pointed to. The smaller of the two elements is output and the pointer in its list is advanced. (The decision as to which to output at a tie is arbitrary.) When one list is used up, simply output the remainder of the other list. The sequence of operations for the lists 1,3,6 and 2,4,5 is shown in Figure 7.1. Since each comparison results in at least one output and the last output is free, we require at most $i + j - 1$ comparisons to merge the lists $u_1 \leq \ldots \leq u_i$ and $v_1 \leq \ldots \leq v_j$. On the other hand, if one list is output before any of the other list, we might use only $\min(i, j)$ comparisons.

Let $\mathtt{C}(n)$ be an upper bound on the number of comparisons needed to merge sort a list of n things. Clearly $\mathtt{C}(1) = 0$. The number of comparisons needed to merge two lists with a total of n items is at most $n - 1$. We can convert our sorting procedure into one for computing $\mathtt{C}(n)$. All we need to do is replace `SORT` with `C` and add up the various counts. Here's the result.

```
Procedure C(n)
    C = 0
    If (n = 1), then Return C
    Let m be n/2 with remainder discarded
    C = C + C(m)
    C = C + C(n − m)
    C = C + (n − 1)
    Return C
End
```

To make things easier for ourselves, let's just look at lengths which are powers of two so that the division comes out even. Then $\mathtt{C}(1) = 0$ and $\mathtt{C}(2^k) = 2\mathtt{C}(2^{k-1}) + 2^k - 1$. By applying the recursion we get

$$\begin{array}{rclcrcl} \mathtt{C}(2) & = & 2 \cdot 0 + 1 & = & 1, & \qquad \mathtt{C}(4) & = & 2 \cdot 1 + 3 & = & 5, \\ \mathtt{C}(8) & = & 2 \cdot 5 + 7 & = & 17, & \qquad \mathtt{C}(16) & = & 2 \cdot 17 + 15 & = & 49. \end{array}$$

What's the pattern?

* * * Stop and think about this! * * *

It appears that $\mathtt{C}(2^k) = (k-1)2^k + 1$. We leave it to you to prove this by induction. This suggests that the number of comparisons needed to merge sort an n long list is bounded by about $n \log_2(n)$. We've only proved this for n a power of 2 and will not give a general proof.

There are a couple of points to notice here. First, we haven't concerned ourselves with how the algorithm will actually be implemented. In particular, we've paid no attention to how storage will be managed. Such a cavalier attitude won't work with a computer so we'll discuss implementation problems in the next section. Second, the recursive algorithm led naturally to a recursive estimate for the speed of the algorithm. This is often true. □

Local Descriptions

We begin with the local description for two ideas we've seen before when discussing decision trees. Then we look at the "Tower of Hanoi" puzzle, using the local description to illustrate the claims for thinking locally made at the beginning of this section.

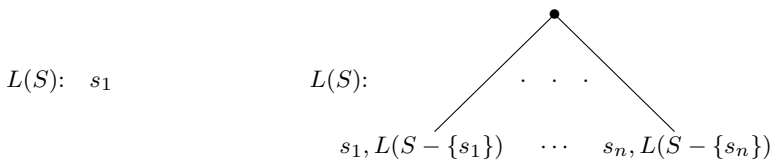


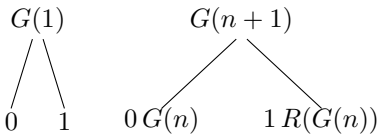
Figure 7.2 The two cases for the local description of $L(S)$, the lex order permutation tree for $S = \{s_1, \ldots, s_n\}$. Left: the initial case $n = 1$. Right: the recursive case $n > 1$.

Example 7.14 The local description of lex order permutations Suppose that S is an n element set with elements $s_1 < \ldots < s_n$. In Section 3.1 we discussed how to create the decision tree for generating the permutations of S in lex order. (See page 70.) Now we'll give a recursive description that follows the pattern in Definition 7.3 (p. 204).

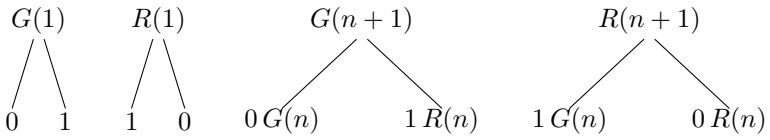
Let $L(S)$ stand for the decision tree whose leaves are labeled with the permutations of S in lex order and whose root is labeled $L(S)$. If x is some string of symbols, let $x, L(S)$ stand for the $L(S)$ with the string of symbols "x," appended to the front of each label of $L(s)$. For Case 1 in Definition 7.3, $n = 1$. Then $L(S)$ is simply one leaf labeled s_1. See Figure 7.2 for $n > 1$,. What we have just given is called the *local description* of the lex order permutation tree because it looks only at what happens from one step of the inductive definition to the next. In other words, a local description is nothing more that the statement of Definition 7.3 for a specific problem.

We'll use induction to prove that this is the correct tree. When $n = 1$, it is clear. Suppose it is true for all S with cardinality less than n. The permutations of S in lex order are those beginning with s_1 followed by those beginning with s_2 and so on. If s_k is removed from those permutations of S beginning with s_k, what remains is the permutations of $S - \{s_k\}$ in lex order. By the induction hypothesis, these are given by $L(S - \{s_k\})$. Note that the validity of our proof does not depend on how they are given by $L(S - \{s_k\})$. □

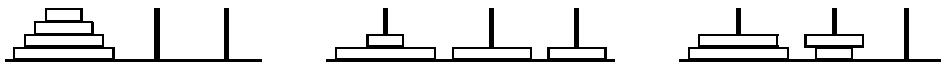
Example 7.15 Local description of Gray code for subsets We studied Gray codes for subsets in Examples 3.12 (p. 82) and 3.13 (p. 86). We can give a local description of the algorithm as



where $n > 0$, $R(T)$ is T with the order of the leaves reversed, and $1T$ is T with 1 prepended to each leaf. Alternatively, we could describe two interrelated trees, where now $R(n)$ is the tree for the Gray code listed in reverse order:



The last tree may appear incorrect, but it is not. When we reverse $G(n + 1)$, we must move $1\,R(n)$ to the left child and reverse it. Since the reversal of $R(n)$ is $G(n)$, this gives us the left child of $G(n + 1)$. The right child is explained similarly. □



(a) Starting (b) Intermediate (c) Illegal!

Figure 7.3 Three positions in the Tower of Hanoi puzzle for $n = 4$.

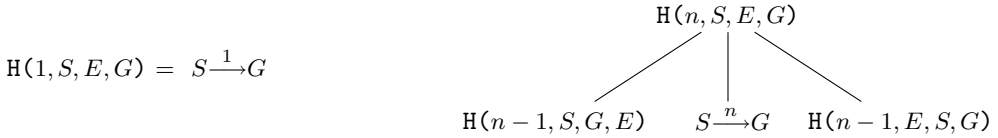


Figure 7.4 The local description of the solution to the Tower of Hanoi puzzle. The left hand figure describes the initial case $n = 1$ and the right hand describes the recursive case $n > 1$. Instead of labeling the tree, we've identified the root vertex with the label. This is convenient if we expand the tree as in the next figure.

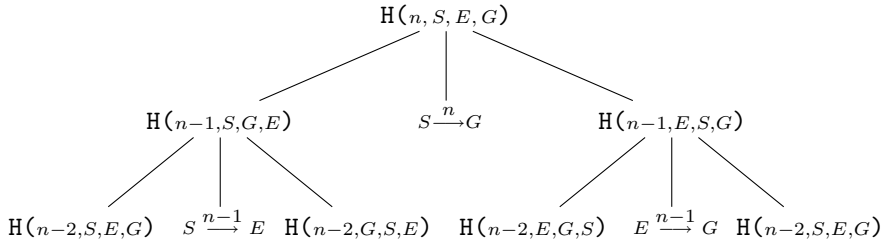


Figure 7.5 The first expansion of the Tower of Hanoi tree for $n > 2$. This was obtained by applying Figure 7.4 to itself.

Example 7.16 The Tower of Hanoi puzzle The *Tower of Hanoi* puzzle consists of n different sized washers (i.e., discs with holes in their centers) and three poles. Initially the washers are stacked on one pole as shown in Figure 7.3(a). The object is to switch all of the washers from the left hand pole to the right hand pole. The center pole is extra, to assist with the transfers. A legal move consists of taking the top washer from a pole and placing on top of the pile on another pole, provided it is not placed on a smaller washer.

How can we solve the puzzle?

To move the largest washer, we must move the other $n - 1$ to the spare peg. After moving the largest, we can then move the other $n - 1$ on top of it. Let the washers be numbered 1 to n from smallest to largest. When we are moving any of the washers 1 through k, we can ignore the presence of all larger washers beneath them. Thus, moving washers 1 through $n - 1$ from one peg to another when washer n is present uses the same moves as moving them when washer n is not present. Since the problem of moving washers 1 through $n-1$ is simpler, we practically have a recursive description of a solution. All that's missing is the observation that the simplest case, $n = 1$, is trivial. The local description of the algorithm is shown in Figure 7.4 where $X \xrightarrow{k} Y$ indicates that washer k is to be moved from peg X to peg Y.

If we want to think globally, we need to expand this tree until all the $H(\cdots)$ are replaced by moves. In other words, we continue until reaching $\mathtt{H}(1, X, Y, Z)$, which is simply $X \xrightarrow{1} Z$. How much expansion is required to reach this state depends on n. The first step in expanding this tree is shown in Figure 7.5.

To get the sequence of moves, expand the tree as far as possible, extend the edges leading to leaves so that they are all on the same level, and then list the leaves as they are encountered reading from left to right. When $n = 3$, we can use Figure 7.5 and the left side of Figure 7.4. The resulting sequence of moves is

$$S \xrightarrow{1} G \quad S \xrightarrow{2} E \quad G \xrightarrow{1} E \quad S \xrightarrow{3} G \quad E \xrightarrow{1} S \quad E \xrightarrow{2} G \quad S \xrightarrow{1} G.$$

As you can see, this is getting rather complex.

Here's some pseudocode that implements the local description. It's constructed directly from Figure 7.4. We execute it by running `H(`n`,start,extra,goal)`.

```
Procedure H(n, S, E, G)
     If (n = 1)
          Print: Move washer 1 from S to G
          Return
     End if
     H(n − 1, S, G, E)
     Print: Move washer n from S to G
     H(n − 1, E, S, G)
End
```

How many moves are required to solve the puzzle? Let the number be h_n. From the local description we have $h_1 = 1$ and $h_n = h_{n-1} + 1 + h_{n-1} = 2h_{n-1} + 1$ for $n > 1$. Using this one can prove that $h_n = 2^n - 1$.

We can prove by induction that the algorithm works. It clearly works for $n = 1$. Suppose $n = 1$. By induction, the left child in the local description (Figure 7.4) moves the $n - 1$ smallest washers from S to E. Thus the move in the middle child is valid. Finally, the right child moves the $n - 1$ smallest washers from E to G (again, by induction).

We can prove other things as well. For example, washer 1 moves on the kth move if and only if k is odd. Again, this is done by induction. It is true for $n = 1$. For $n > 1$, it true for the left child of local description by induction. Similarly, it is true for the right child because the left child and the middle child involve a total of $h_{n-1} + 1 = 2^{n-1}$ moves, which is an even number. If you enjoy challenges, you may wish to pursue this further and try to determine for all k when washer k moves as well as what each move is. Determining "when" is not too difficult. Determining "what" is tricky. □

*Computer Implementation

Computer implementation of recursive procedures involves the use of stacks to store information for different levels of the recursion. A *stack* is a list of data in which data is added to the end of the list and is also removed from the same end. The end of the list is called the top of the stack, adding something is called *pushing* and removing something is called *popping*.

Example 7.17 Implementing the Tower of Hanoi solution Let's return to the Tower of Hanoi procedure `H`(n, S, E, G), which is described in Figure 7.4. To begin, we push n, S, E and G on the stack and call the program `H`. The stack entries, from the top, may be referred to as the first, second and so on items. If $n = 1$, `H` simply carries out the action in the left side of Figure 7.4. If $n > 1$, it carries out actions corresponding to each of the three sons on the right side of Figure 7.4 in turn: The left son causes it

- to push $n - 1$ and the second, fourth and third items on the stack, in that order,
- to call the program `H`

and, when `H` finishes,

- to pop four items off the stack.

The middle son is similar to $n = 1$ and the right son is similar to the left son.

You may find it helpful to see what this process leads to when $n = 3$. How does it relate to Figure 7.5? □

Example 7.18 Computing a bound on comparisons in merge sorting Let's look at the pseudocode for computing `C`(n), the upper bound on the number of comparisons in our merge sort (Example 7.13 (p. 211)). Here it is

```
Procedure C(n)
     C = 0
     If (n = 1),
          Return C
     End if
     Let m be n/2 with remainder discarded
     C = C + C(m)
     C = C + C(n − m)
     C = C + (n − 1)
     Return C
End
```

This has a new feature not present in `H`: The procedure contains the variable C which we must save if we call `C` recursively. This can be done as follows. When a procedure is called, space is allocated on (pushed onto) the stack to store all of its "local variables" (that is, variables that exist only in the procedure). When the procedure is done the space is deallocated (popped off the stack). Thus, in a programming language that permits recursion, each call of a procedure `Proc` uses space for

- the address in the calling procedure to return to when `Proc` is done,
- the values of the variables passed to `Proc` and
- the values of the variables that are local to `Proc`

until the procedure `Proc` is done. Since a recursive procedure may call itself, which calls itself, which calls itself, ... ; it may use a considerable amount of storage. □

Example 7.19 Implementing merge sorting Look at example Example 7.13 (p. 211). It requires a tremendous amount of extra storage since we need space for the s, t, u and v arrays every time the procedure calls itself. If we want to implement this algorithm on a real computer, it will have to be rewritten to avoid creating arrays recursively. This can be done by placing the sorted array in the original array. Here's the new version

```
Procedure SORT(a[lo] through a[hi])
    If (lo = hi), then Return
    Let m be (lo + hi)/2 with remainder discarded
    SORT(a[lo] through a[m])
    SORT(a[m + 1] through a[hi])
    MERGE(a[lo] through a[m] with a[m + 1] through a[hi])
End
```

This requires much less storage. A simple implementation of `MERGE` requires a temporary array, but multiple copies of that array will not be created through recursive calls because `MERGE` is not recursive. The additional array problem can be alleviated. We won't discuss that. □

Exercises

7.3.1. In Example 7.13 we computed an upper bound $C(n)$ on the number of comparisons required in a merge sort. The purpose of this exercise is to compute a lower bound. Call this bound $c(n)$.

(a) Explain why merging two sorted lists of lengths k_1 and k_2 requires at least $\min(k_1, k_2)$ comparisons, where "min" denotes minimum. Give an example of when this is achieved for all values of k_1 and k_2.

(b) Write code like Procedure C(n) in Example 7.13 to compute $c(n)$.

(c) State and prove a formula for $c(n)$ when $n = 2^k$, a power of 2. Compare $c(n)$ with $C(n)$ when n is a large power of 2.

7.3.2. Give a local description of listing the strictly decreasing functions from $\underline{k}$ to $\underline{n}$ in lex order. (These are the k-subsets of $\underline{n}$.) Call the list $D(n, k)$ and use the notation $i, D(j, k)$ to mean the list obtained by prepending i to each of the functions in $D(j, k)$ written in one-line form. For example

$$D(3,2) \;=\; (2,1;\;\; 3,1;\;\; 3,2) \qquad \text{and} \qquad 5, D(3,2) \;=\; (5,2,1;\;\; 5,3,1;\;\; 5,3,2).$$

7.3.3. Merging two lists in a single list stored elsewhere requires that each item be moved once. Dividing a list approximately in two requires no moves. State and prove a formula for the number of moves required by a merge sort of n items when $n = 2^k$, a power of 2.

7.3.4. We have a pile of n coins, all of which are identical except for a single counterfeit coin which is lighter than the other coins. We have a "beam balance," a device which compares two piles of coins and tells which pile is heavier. Here is a recursive algorithm for finding the counterfeit coin in a set of $n \geq 2$ coins.

```
Procedure Find(n,Coins)
     If (n = 2) Put one coin in each pile
     and report the result.
     Else
          Select a coin C in Coins.
          Find(n − 1,Coins−C)
          If a counterfeit is reported, report it.
          Else report C.
          Endif
     Endif
End
```

Since `Find` only uses the beam balance if $n = 2$, this recursive algorithm finds the counterfeit coin by using the beam balance only once regardless of the value of $n \geq 2$. What is wrong? How can it be corrected?

7.3.5. Suppose we have a way to print out the characters 0–9 and $-$, but do not have a way to print out integers such as -360. We want a procedure `OUT`(m) to print out integers m, both positive, negative, and zero, as strings of digits. If $n \geq 0$ is a positive integer, let q and r be the quotient and remainder when n is divided by 10.

(a) Using the fact the digits of n are the digits of q followed by the digit r to write a *recursive* procedure `OUT`(m) that prints out m for any integer (positive, negative, or zero).

(b) What are the simplest objects in your recursive solution?

(c) Explain why your procedure never runs forever.

7.3.6. Let $n \geq 0$ be an integer and let q and r be the quotient and remainder when n is divided by 10. We want a procedure `DSUM`(n) to sum the digits of n.

(a) Using the fact that the sum of the digits of n equals r plus the sum of the digits of q, write a recursive procedure `DSUM`(n).

(b) What are the simplest objects in your recursive solution?

(c) Explain why your procedure never runs forever.

7.3.7. What is the local description for the tree that generates the decreasing functions in $\underline{n^k}$? Decreasing functions were discussed in Example 3.8.

7.3.8. Expand the local description of the Tower of Hanoi to the full tree for $n = 2$ and for $n = 4$. Using the expanded trees, write down the sequence of moves for $n = 2$ and for $n = 4$.

7.3.9. Let $S(n)$ be the number of moves required to solve the Tower of Hanoi puzzle.

(a) Prove by induction that `Procedure H` takes the least number of moves.

(b) Convert `Procedure H` into a procedure that computes $S(n)$ recursively as was done for sorting in Example 7.13. Translate the code you have just written into a recursion for $S(n)$.

(c) Construct a table of $S(n)$ for $1 \leq n \leq 7$.

(d) Find a simple formula (*not* a recursion) for $S(n)$ and prove that it is correct by using the result in (b) and induction.

(e) Assuming the starting move is called move one, what washer is moved on move k?
Hint. There is a simple description in terms of the binary representation of k.

*(f) What are the source and destination poles on move k?

7.3.10. We have discovered a simpler procedure for the Tower of Hanoi: it only involves one recursive call. To move washers k to n we use $\texttt{H}(k, n, S, E, G)$. Here's the procedure.

```
Procedure H(k, n, S, E, G)
    If (k = n)
        Move  washer n from S to G
        Return
    End if
    Move  washer k from S to E
    H(k + 1, n, S, E, G)
    Move  washer k from E to G
    Return
End
```

To get the solution, run $\texttt{H}(1, n, S, E, G)$. This is an incorrect solution to the Tower of Hanoi problem. Which of the two conditions for a recursive solution fails and how does it fail? Why does the algorithm in the text not fail in the same way?

7.3.11. We consider a modification of the Tower of Hanoi. All the old rules apply, but the moves are more limited: You can think of the poles as being in a row with the extra pole in the middle. The new rule then says that a washer can only move to an adjacent pole. In other words, a washer can never be moved directly from the original starting pole to the original destination pole. Thus, when $n = 1$ we require two moves: $S \xrightarrow{1} E$ and $E \xrightarrow{1} G$.

Let $H^*(n, P_1, P_2, P_3)$ be the tree that moves n washers from P_1 to P_3 while using P_2 as the extra pole. The middle pole is P_2.

(a) At the start of the problem, we described the moves for $n = 1$. For $n > 1$, washer n must first move to the extra post and then to the goal. The other $n - 1$ washers must first be stacked on the goal and then on the start to allow these moves. Draw the local description of H^* for $n > 1$.

(b) Let h_n^* be the number of washers moved by $H^*(n, S, E, G)$. Write down a recursion for h_n^*, including initial conditions.

(c) Compute the first few values of h_n^*, guess the general solution, and prove it.

7.3.12. The number of partitions of the set $\underline{n}$ into k blocks was defined in Example 1.27 to be $S(n, k)$, the Stirling numbers of the second kind. We developed the recursion $S(n, k) = S(n-1, k-1)+kS(n-1, k)$ by considering whether n was in a block by itself or in one of the k blocks of $S(n - 1, k)$. By using the actual partitions instead of just counting them, we can interpret the recursion as a means of producing all partitions of $\underline{n}$ with k blocks.

(a) Write pseudocode to do this.

(b) Draw the local description for the algorithm.

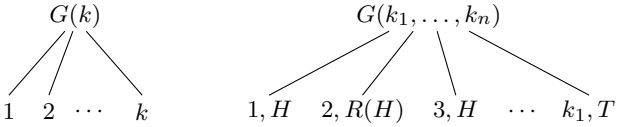
7.3.13. We want to produce all sequences $\alpha = a_1, \ldots, a_n$ where $1 \le a_i \le k_i$. This is to be done so that if β is produced immediately after α, then all but one of the entries in β is the same as in α and that entry differs from the α entry by one. Such a list of sequences is called a *Gray code*. If T is a tree whose leaves are labeled by sequences, let a, T be the same tree with each leaf label α replaced by a, α. Let $R(T)$ be the tree obtained by taking the mirror image of T. (The sequences labeling the leaves are moved but are not reversed.) For example, if the leaves of T are labeled, from left to right,

$$1,2 \qquad 1,3 \qquad 2,4 \qquad 1,4 \qquad 2,3,$$

then the leaves of $R(T)$ are labeled, from left to right,

$$2,3 \qquad 1,4 \qquad 2,4 \qquad 1,3 \qquad 1,2.$$

Let $G(k_1, \ldots, k_n)$ be the decision tree with the local description shown below. Here $n > 1$, $H = G(k_2, \ldots, k_n)$ and T is either H or $R(H)$ according as k_1 is odd or even.



(a) Draw the full tree for $G(3,2,3)$ and the full tree for $G(2,3,3)$.

(b) Prove that $G(k_1, \ldots, k_n)$ contains all sequences $a_1, \ldots, a_n$ where $1 \le a_i \le k_i$.

(c) Prove that adjacent leaves of $G(k_1, \ldots, k_n)$ differ in exactly one entry and that entry changes by one from one leaf to the next.

*(d) Suppose that $k_1 = \cdots = k_n = 2$. Describe RANK(α).

*(e) Suppose that $k_1 = \cdots = k_n = 2$. Tell how to find the sequence that follows α without using RANK and UNRANK.

7.3.14. For each of the previous exercises that requested pseudocode, tell what is placed on the stack as a result of the recursive call.

7.4 Divide and Conquer

In its narrowest sense, divide and conquer refers to the division of a problem into a few smaller problems that are of the same kind as the original problem and so can be handled by a recursive method. We've seen binary insertion, Quicksort and merge sorting as examples of this. In a broader sense, divide and conquer refers to any method that divides a problem into a few simpler problems. Heapsort illustrates this broader definition.

The broad divide and conquer technique is important for dealing with most complex situations. Delegation of responsibility is an application in everyday life. Scientific investigation often employs divide and conquer. In computer science it appears in both the design and implementation of algorithms, where it is referred to by such terms as "top-down programming," "structured programming," "object oriented programming" and "modularity." Properly used, these are techniques for efficiently creating and implementing understandable, correct and flexible programs.

What tools are available for applying divide and conquer to smaller problems? For example, how might one discover the algorithms we've discussed in this text? An algorithm that has a nonrecursive nature doesn't seem to fit any general rules; for example, all we can recommend for discovering something like Heapsort is inspiration. You can cultivate inspiration by being familiar with a variety of ideas and by trying to look at problems in novel ways.

We can say a bit more about trying to discover recursive algorithms; that is, algorithms that we use divide and conquer in its narrowest sense. Suppose the data is such that it is possible to split it into a few large blocks. You can ask yourself if anything is accomplished by solving the original

problem on the separate blocks and then exploiting it. We'll see how this works by first reviewing earlier material and then move on to some new problems.

Example 7.20 Locating items in lists If we are trying to find an item in an ordered list, we can divide the list in half. What does this accomplish?

Suppose the item actually lies in the first half. When we look at the start of the second half list, we can immediately solve the problem for that sublist: the item we're looking for is not in that sublist because it precedes the first item. Thus we've divided the original problem into two problems, one of which is trivial. Binary insertion exploits this observation. Analysis of the resulting algorithm shows that it is considerably faster than simply looking through the list item by item. □

Example 7.21 Sorting If we are trying to sort a list, we could divide it into two parts and sort each part separately. What does this accomplish? That depends on how we divided the list.

Suppose that we divided it in half arbitrarily. If each half is sorted, then we must merge two sorted lists. Some thought reveals that this is a fairly easy process. Exploiting this idea leads to merge sorting. Analysis of the algorithm shows that it is fast.

Suppose that we can arrange the division so that all the items in the first part should precede all the items in the other part. When the two parts are sorted, the list will be sorted. How can we divide the list this way? A bit of thought and a *clever idea* may lead to the method used by Quicksort. Analysis of the algorithm shows that it is usually fast. □

Example 7.22 Calculating powers Suppose that we want to calculate x^n when n is a large positive integer. A simple way to do this is to multiply x by itself the appropriate number of times. This requires $n-1$ multiplications.

We can do better with divide and conquer. Suppose that $n = mk$. We can compute $y = x^m$ and then compute x^n by noting that it equals y^k. Using the method in the previous paragraph to compute y and then to compute y^k means that we require only $(m-1)+(k-1) = m+k-2$ multiplications. This is much less than $n-1 = mk-1$. We'll call this the "factoring method."

As is usual with divide and conquer, recursive application of the idea is even better.

In other words, we regard the computation of x^m and y^k as new problems and solve them by first factoring m and k and so forth. For example, computing x^{2^t} requires only t multiplications.

There is a serious drawback with the factoring method: n may not have many factors; in fact, it might even be a prime. What can we do about this?

If $n > 3$ is a prime, then $n-1$ is not a prime since it is even. Thus we can use the factoring method to compute x^{n-1} and then multiply it by x. We still have to deal with the factorization problem. This is getting complicated, so perhaps we should look for a simpler method. Try to think of something.

* * * Stop and think about this! * * *

The request that you try to think of something was quite vague, so it is quite likely that different people would have come up with different ideas. Here's a fairly simple method that is based on the observations $x^{2m} = (x^m)^2$ and $x^{2m+1} = (x^m)^2 x$ applied recursively. Let the binary representation of n be $b_k b_{k-1} \dots b_0$; that is

$$n = \sum_{i=0}^{k} b_i 2^i = \Big(\cdots\big((b_k)2 + b_{k-1}\big)2 + \cdots b_1\Big)2 + b_0. \tag{7.10}$$

It follows that

$$x^n = \Big(\cdots\big((x^{b_k})^2 x^{b_{k-1}}\big)^2 \cdots x^{b_1}\Big)^2 x^{b_0},$$

where x^{b_i} is either x or 1 and so requires no multiplications to compute. Since multiplication by 1 is trivial, the total number of multiplications is k (from squarings) plus the number of b_0 through b_{k-1}

which are nonzero. Thus the number of multiplications is between k and $2k$. Since $2^{k+1} > n \geq 2^k$ by (7.10), this method always requires $\Theta(\ln n)$ multiplications.[1] In contrast, our previous methods always required at least $\Theta(\ln n)$ multiplications and sometimes required as many as $\Theta(n)$.

Does our latest method require the least number of multiplications? Not always. There is no known good way to calculate the minimum number of multiplications required to compute x^n. □

Example 7.23 Finding a maximum subsequence sum Suppose we are given a sequence of n arbitrary real numbers $a_1, a_2, \ldots, a_n$. We want to find i and $j \geq i$ such that $\sum_{k=i}^{j} a_k$ is a large as possible.

Here's a simple approach: For each $1 \leq i \leq j \leq n$ compute $A_{i,j} = \sum_{k=i}^{j} a_k$, then find the maximum of these numbers. Since it takes $j - i$ additions to compute $A_{i,j}$, the number of additions required is

$$\sum_{j=1}^{n} \sum_{i=1}^{j} (j - i),$$

which turns out to be approximately $n^3/6$. The simple approach to find the maximum of the approximately $n^2/2$ numbers $A_{i,j}$ requires about $n^2/2$ comparisons. Thus the total work is $\Theta(n^3)$.

Can we do better? Yes, there are ways to compute the $A_{i,j}$ in $\Theta(n^2)$. The work will be $\Theta(n^2)$.

Can we do better? Yes, there is a divide and conquer approach. The idea is

- split $a_1, \ldots, a_n$ into two sequences $a_1, \ldots, a_k$ and $a_{k+1}, \ldots, a_n$, where $k \approx n/2$,
- compute the information for the two halves recursively,
- put the two halves together to get the information for the original sequence $a_1, \ldots, a_n$.

There is a problem with this. Consider the sequence $3, -4, 2, 2, -4, 3$ the two half sequences each have a maximum of 3, but the maximum of the entire sequence is $2 + 2 = 4$. The problem arises because the maximum sum is split between the two half-sequences $3, -4, 2$ and $2, -4, 3$. We get around this by keeping track of more information. In fact we keep track of the maximum sum, the maximum sum that starts at the left end of the sequence, the maximum sum that ends at the right end of the sequence, and the total sum. Here's an algorithm.

```
Procedure MaxSum(M, L, R, T, (a_1, ..., a_n))
    If n = 1
        Set M = L = R = T = a_1
    Else
        k = ⌊n/2⌋
        MaxSum(M_ℓ, L_ℓ, R_el, T_ℓ, (a_1, ..., a_k))
        MaxSum(M_r, L_r, R_r, T_r, (a_{k+1}, ..., a_n))
        M = max(M_ℓ, M_r, R_ℓ + L_r)
        L = max(L_ℓ, T_ℓ + L_r)
        R = max(R_r, T_r + R_ℓ)
        T = T_ℓ + T_r
    End if
    Return
End
```

[1] The notation Θ indicates same rate of growth to within a constant factor. For more details, see page 368.

Why does this work?

- It should be clear that the calculation of T is correct.
- You should be able to see that M, the maximum sum, is either the maximum in the left half (M_ℓ), the maximum in the right half (M_r), or a combination from each half ($R_\ell + L_r$). This is just what the procedure computes.
- L, the maximum that starts on the left, either ends in the left half (L_ℓ) or extends into the right half ($T_\ell + L_r$), which is what the procedure computes.
- The reasoning for R is the mirror image of that for L.

How long does this algorithm take to run? Ignoring the recursive part, there is a constant amount of work in the code, with one constant for $n = 1$ and another for $n > 1$. Hence

$$\text{(total time)} \quad = \quad \Theta(1) \times \text{(number of calls of MaxSum)}.$$

Every call of MaxSum when $n > 1$ divides the sequence. We must insert $n - 1$ divisions between the elements of $a_1, \ldots, a_n$ to get sequences of length 1. Hence there are $n - 1$ calls of this type. `MaxSum` also calls itself for each of the n elements of the sequence. Thus there are a total of $2n - 1$ calls and so the running time is $\Theta(n)$. □

There is an important principle that surfaced in the previous example which didn't arise in our simpler examples of finding a recursive algorithm:

Principle *In order to find a recursive algorithm for a problem, it may be helpful, even necessary, to ask for more—either a stronger result or the calculation of more information.*

In the example, we introduced new variables in our algorithm to keep track of other sums. Without such variables, the algorithm would not have worked. As we remarked in Section 7.1, this principle also applies to inductive proofs. We have seen some examples of this:

- When we proved the existence of lineal spanning trees in Theorem 6.2 (p. 153), it was necessary to prove that we could find one with any vertex of the graph as the root of the tree.
- When we studied Ramsey problems in Example 7.4 (p. 201), we had to replace $N(k)$ with the more general $N(k_1, k_2)$ in order to carry out our inductive proof.

Exercises

7.4.1. What is the least number of multiplications you can find to compute each of the following: x^{15}, y^{21}, z^{47} and w^{49}.

7.4.2. This problem concerns the Fibonacci numbers. They satisfy the recursion $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. People use various values for F_0 and F_1. We will use $F_0 = 0$ and $F_1 = 1$. (Elsewhere in the text we may find it convenient to choose different initial values.)

(a) Compute F_n for $n \leq 7$.

(b) Recall that A^{t} means the transpose of the matrix A. Let $\vec{v}_n$ be the column vector $(F_n, F_{n+1})^{\mathrm{t}}$. Show that $\vec{v}_{n+1} = M\vec{v}_n$ where $M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Conclude that $\vec{v}_n = M^n\vec{v}_0$. Suggest a rapid method for calculating F_n.

(c) Show that

$$M^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix}.$$

(d) Use $M^{2n} = (M^n)^2$ to prove that $F_{2n} = F_n(F_{n+1} + F_{n-1}) = F_{n+1}^2 - F_{n-1}^2$ and $F_{2n+1} = F_{n+1}^2 + F_n^2$.

7.4.3. Suppose that $a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k}$ for $n \geq k$. Extend the idea in the previous exercise to describe how to rapidly compute a_n for a large value of n.

7.4.4. In this problem you are given a bag of n coins. All the coins have the same weight, except for one, which is counterfeit. You are given a balance type scales. This means that given two piles of coins, you can use the scales to determine whether the piles have the same weight or, if they differ, which is heavier. The goal is to devise a strategy for locating the counterfeit coin with the least possible number of weighings. (See Exercise 7.3.4 (p. 218).)

(a) Given that the counterfeit coin is lighter, devise a recursive divide and conquer strategy for finding it.
Hint. If you strategy is a good one, it will find the coin after k weighings when $n = 3^k$.

(b) Modify the proof of the lower bound for the number of comparisons needed for sorting to show that the average number of weighings needed in any algorithm is at least $\log_3 n$.

(c) Devise an algorithm when it is not known whether the counterfeit coin is lighter or heavier.

(d) Suppose that there are two counterfeit coins, both of the same weight and both lighter than the real coins. Devise a strategy.

*7.4.5. A full binary RP-tree is a rooted plane tree in which each vertex is either a leaf or has exactly two children. An example is a decision tree in which each decision is either "yes" or "no".

Suppose that we are given a full binary RP-tree T with root r and a function f from vertices to the real numbers. If v is a vertex of T, let $F(v)$ be the sum of $f(x)$ over all vertices x in the subtree rooted at v. We want to find

$$F(T) = \max_{v \in T} F(v).$$

One way to do this is compute $F(v)$ for each vertex $v \in T$ and then find the maximum. This takes $\Theta(n \ln n)$ work, where n is the number of vertices in T. Find a better method.

*7.4.6. If you found Example 7.23 easy, here's a challenge: Extend the problem to a doubly indexed array $a_{k,m}$. In this case we want i_1, i_2, j_1 and j_2 so that $\sum_{k=i_1}^{j_1} \sum_{m=i_2}^{j_2} a_{k,m}$ is as large as possible. We warn you that you must keep track of quite a bit more information.

7.4.7. Here's another approach to the identities of Exercise 7.4.2. Let $a_n = a_{n-1} + a_{n-2}$, where a_0 and a_1 are given.

(a) Prove that $a_n = a_0 F_{n-1} + a_1 F_n$ for $n > 0$.

(b) Show that, if $a_0 = F_k$ and $a_1 = F_{k+1}$, then $a_n = F_{n+k}$. Conclude that $F_{n+k} = F_k F_{n-1} + F_{k+1} F_{n-2}$.

Notes and References

Most introductory combinatorics texts discuss induction, but discussions of recursive algorithms are harder to find. The subject of recursion in treated beautifully by Roberts [4]. Williamson [6; Ch.6] discusses the basic ideas behind recursive algorithms with applications to graphs.

Further discussion of Ramsey's Theorem as well as some of its consequences appears in Cohen's text [2; Ch. 5]. A more advanced treatment of the theorem and its generalizations has been given in the monograph by Graham, Rothschild and Spencer [3].

People have studied the Tower of Hanoi puzzle with the same rules but with more than one extra pole for a total of $k > 3$ poles. The optimal strategy is not known; however, it is known that if $H_n(k)$ is the least number of moves required, then $\log_2 H_n(k) \sim (n(k-2)!)^{1/(k-2)}$ [1].

Divide and conquer methods are discussed in books that teach algorithm *design*; however, little has been written on general design strategies. Some of our discussion is based on the article by Smith [5].

1. Xiao Chen and Jian Shen, On the Frame-Stewart conjecture about the Towers of Hanoi, *SIAM J. Comput.* **33** (2004) 584–589.
2. Daniel I.A. Cohen, *Basic Techniques of Combinatorial Theory*, John Wiley (1978).
3. Ronald L. Graham, Bruce L. Rothschild and Joel H. Spencer, *Ramsey Theory*, 2nded., John Wiley (1990).
4. Eric Roberts, *Thinking Recursively*, John Wiley (1986).
5. Douglas R. Smith, Applications of a strategy for designing divide-and-conquer algorithms, *Science of Computer Programming* **8** (1987), 213–229.
6. S. Gill Williamson, *Combinatorics for Computer Science*, Dover (2002).

CHAPTER 8

Sorting Theory

Introduction

The problem of developing and implementing good sorting algorithms has been extensively studied. If you've taken a programming course, you have probably seen code for specific sorting algorithms. You may have programmed various sorting algorithms. Our focus will be different, emphasizing the general framework over the specific implementations. We'll also look at "sorting networks" which are a type of hardware implementation of certain sorting algorithms.

In the last section, we'll explore the "divide and conquer" technique. The major aspect of this technique is the recursive approach to problems.

Before discussing sorting methods, we'll need a general framework for thinking about the subject. Thus we'll look briefly at how sorting algorithms can be classified. All of them involve making comparisons, require some sort of storage medium for the list and must be physically implemented in some fashion. We can partially classify algorithms according to how these things are handled.

- Type of comparison:
  - Relative: An item is compared to another item. The result simply says which of the two is larger. Almost all sorts are of this type.
  - Absolute: An item is ranked using an absolute standard. The result says which of several items it equals (or lies between). These are the *bucket* sorts.
- Data access needed:
  - Random Access: These types of sorts need to be able to look at almost any item in the (partially sorted) list at almost any time.
  - Sequential Access: These types of sorts make a rather limited number of sequential passes through the data. Consequently, the data could be stored on *magnetic tape*.[1] These are usually *merge* sorts.
- Implementation method:
  - Software: Most sorts are implemented as a program on a general purpose computer.

[1] Magnetic tape is a medium that was used before large disc drives were available at reasonable prices. The data on the tape can be thought of as one long list. It was very time consuming to move to a position in the list that was far away from the current position. Hence it was desirable to read the list without skipping around. This is the origin of the term *sequential access*. One tape provided a single sequential access medium. Two tapes provided two sequential access media, and so forth.

- Hardware: Some sorts are implemented by hardware. Depending on how versatile the hardware is, it could be close to a software implementation. The least versatile (and also most common) hardware implementations are (a) the card sorters of a largely bygone era which are used to bucket sort punched cards and (b) *sorting networks*, which we'll study in Section 8.3.

8.1 Limits on Speed

Suppose someone comes to you with two sorting algorithms and asks you which is faster. How do you answer him? Unless he has actual code for a particular machine, you couldn't obtain actual times. However, since comparisons are the heart of sorting, we could ask: "How many comparisons does this algorithm make in the process of sorting?" We could then suggest that the algorithm that required less comparisons was the faster of the two. There are some factors that make this only a rough estimate:

- We did not include relocation overhead—somehow items must be repositioned to obtain the sorted list.
- We did not include miscellaneous overhead such as initialization and subroutine calls.

We will ignore such problems and just look at the number of comparisons. Even so, there are problems:

- Using the parallel processing capabilities of supercomputers or special purpose devices will throw time estimates off because more than one comparison can be done at a time. The amount of parallelism that is possible can vary from algorithm to algorithm.
- The number of comparisons needed may vary greatly, depending on the order of the items in the unsorted list.

We'll ignore these factors in the discussion, except for parallelism in sorting networks, where it is of major importance.

Besides all these problems with estimating running time, there is another problem: Running time is not the only standard that can be used to decide how good an algorithm is. Other important questions include

- How long will it take to get an error free program running?
- How much storage space will the algorithm require?

We'll ignore these issues and focus on running time.

Let $C(n)$ be the number of comparisons that an algorithm requires to sort n items. "Foul!" cries a careful reader. "You pointed out that the time a sort takes depends on the original order of the list and now you're talking about this number $C(n)$ as if that weren't the case." True. We should specify $C(n)$ more carefully. It's reasonable to consider two measures of speed:

- Worst case: $C(n) = \mathrm{WC}(n)$, the greatest number of comparisons the algorithm requires to sort n items. This is important where the results of a sort are needed quickly.
- Average: $C(n) = \mathrm{AC}(n)$, the average number of comparisons the algorithm requires to sort n items. This is important when we are doing many sorts and want to minimize overall computer usage.

The average referred to here is the average over all $n!$ possible orderings of the list. Obviously $\mathrm{WC}(n) \geq \mathrm{AC}(n)$. Our goal in this section is to motivate and prove

Theorem 8.1 Lower Bound on Comparisons *$\mathrm{AC}(n)$, the average number of comparisons required by any sorting algorithm that correctly sorts all possible lists of n items by comparing pairs of elements is at least $\log_2(n!)$.*

By Stirling's formula (Theorem 1.5 (p. 12)), this bound is close to $n \log_2 n$ when n is large. In view of this, a sorting algorithm with running time $\Theta(n \ln n)$ is usually considered to be a reasonably fast algorithm. Most commonly used sorting algorithms are reasonably fast. We'll look at an old friend now.

Example 8.1 Merge sorting is reasonably fast In Example 7.13 (p. 211) we saw that a simple merge sort takes at most about $n \log_2 n$ comparisons when n is a power of 2. (Actually, this result is true for all large n.) Thus, a merge sort is reasonably fast. As indicated in Example 7.13, there are some storage space problems with merge sorting. □

Motivation and Proof of the Theorem

Our proof of Theorem 8.1 will be by induction. Induction requires knowing the result (in this case $\mathrm{AC}(n) \geq \log_2(n!)$) beforehand. How would one ever come up with the result beforehand? A result like the Four Color Theorem (p. 158) might be conjectured after some experimentation, but one is unlikely to stumble upon Theorem 8.1 experimentally. We will "discover" it by relating sorting algorithms to decision trees, which lead easily to the inequality $\mathrm{WC}(n) \geq \log_2(n!)$. One might then test $\mathrm{AC}(n) \geq \log_2(n!)$ for small values of n, thus motivating Theorem 8.1. Someone versed in information theory might motivate the theorem as follows: "A comparison gives us one bit of information. Given k bits of information, we can distinguish among at most 2^k different things. Since we must distinguish among $n!$ different arrangements, we require that $2^k \geq n!$ and so $k \geq \log_2(n!)$." (This is motivation, not a proof—it's not even clear if it's referring to worst case or average case behavior.) Let's get on with things.

Suppose we are given a comparison based sorting algorithm. Since it is assumed to correctly sort n-lists, it must correctly sort lists in which all items are different. By simply renaming our items 1, 2, ..., n, we can suppose that we are sorting lists which are permutations of $\underline{n}$.

Our proof will make use of the decision tree associated with this sorting algorithm. We construct the tree as follows. Whenever we make a comparison in the course of the algorithm, our subsequent action depends on whether an inequality holds or fails to hold. Thus there are two possible decisions at each comparison and so each vertex in our decision tree has at most two sons.

Label each leaf of the tree with the permutations that led to that leaf and throw away any leaves that are not associated with any permutation. To do this, we start at the root with a permutation f of $\underline{n}$ and at each vertex in the tree we go left if the inequality we are checking holds and go right if it fails to hold. At the same time, we carry out whatever manipulations on the data in f that the algorithm requires. When we arrive at a leaf, the data in f will be sorted. Label the leaf with the f we started out with at the root, written in one line form. Do this for all $n!$ permutations of $\underline{n}$.

For example, consider the following algorithm for sorting a permutation of $\underline{3}$.

1. If the entry in the first position exceeds the entry in the third position, switch them.
2. If the entry in the first position exceeds the entry in the second position, switch them.
3. If the entry in the second position exceeds the entry in the third position, switch them.

Figure 8.1 shows the labeled decision tree. Two positions where you would ordinarily expect leaves have none because they are never reached. Consider the permuted sequence 231. The "if" in Step 1 is true and results in a switch to give 132. The second "if" is false and results in no switch. The third

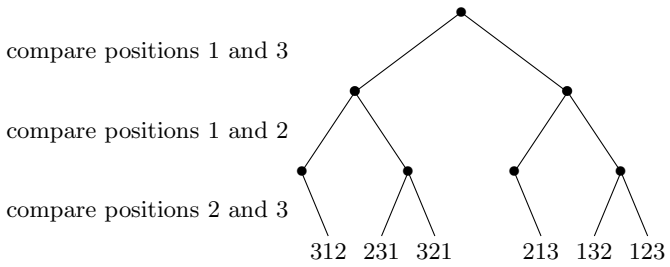


Figure 8.1 A sorting algorithm decision tree. Leftward branches correspond to decisions to switch. Leaves are labeled with starting sequences.

is true and results in a switch to give 123. Thus, the sequence of decisions associated with sorting 231 is switch, no switch and switch, respectively.

We now show that a decision tree for a correct sorting algorithm has exactly $n!$ leaves. This will follow from the fact that each leaf is labeled with exactly one permutation. Why can't we have two permutations at the same leaf? Since each leaf in any such decision tree is associated with a particular rearrangement of the data, both permutations of $\underline{n}$ would be rearranged in the same fashion. Since they differed originally, their rearrangements would differ. Thus at least one of the rearrangements would not be $1, 2, \ldots, n$ and so would be incorrectly sorted.

These decision trees are binary RP-trees. (An RP-tree was defined in Definition 5.12 (p. 139) and it is binary if each node has at most two sons.) The set B of all binary RP-trees with $n!$ leaves includes the set S of those decision trees that come from sorting algorithms. Since S is a subset of B, a lower bound for any function on B is also a lower bound for that function on S. Hence a lower bound on worst case or average values for the set of all binary RP-trees with $n!$ leaves will also be a lower bound on $\mathrm{WC}(n)$ or $\mathrm{AC}(n)$ for any algorithm that correctly sorts n items by using pairwise comparisons.

In our decision tree approach, a comparison translates into a decision. If we only wanted to study $\mathrm{WC}(n)$, we could finish quickly as follows. By the definition of $\mathrm{WC}(n)$, the longest possible sequence of decisions contains $\mathrm{WC}(n)$ decisions. This is called the *height* of the tree. To get a tree with as many leaves as possible, we should let the tree branch as much as possible. Since the number of nodes doubles for each level of decisions in such a tree, you should be able to see that a binary RP-tree of height k has at most 2^k leaves. Since there are $n!$ leaves, we must have $2^{\mathrm{WC}(n)} \geq n!$. Taking $\log_2$ of both sides gives us $\mathrm{WC}(n) \geq \log_2(n!)$.

What about $\mathrm{AC}(n)$? It's not too difficult to compute the average number of comparisons for various binary RP-trees when they are not too large. If you were to do that for a while, you would probably begin to believe that the lower bound we just derived for $\mathrm{WC}(n)$ is also a lower bound for $\mathrm{AC}(n)$, as claimed in the theorem. This completes the motivation for believing the theorem.

How might we prove the theorem? Since the first decision (the one at the root) divides the tree into two smaller trees, it seems reasonable to try induction. Unfortunately, a tree with $(n+1)!$ leaves is a lot bigger than one with only $n!$ leaves. This can cause problems with induction. Let's sever our ties with sorting algorithms and consider decision trees with any number of leaves, not just those where the number of leaves is $n!$ for some n.

From now on, n will now indicate the total number of leaves in the tree, not the number of things being permuted.

For a decision tree T, let $\mathrm{TC}(T)$ be the sum, over all leaves ℓ of T, of the number of decisions needed to reach ℓ. The average cost is then $\mathrm{TC}(T)/n$. To prove the theorem, it suffices to prove

$$\mathrm{TC}(T) \geq n \log_2 n \text{ for all binary RP-trees } T \text{ with } n \text{ leaves.} \qquad 8.1$$

Call this $\mathcal{A}(n)$. Clearly $\mathcal{A}(1)$ is true since $\log_2 1 = 0$.

We now proceed by induction: Given $\mathcal{A}(k)$ for all $k < n$, we will prove $\mathcal{A}(n)$. Let T be a binary RP-tree with n leaves. Consider the root of T. If it has only one son, removing the root gives a binary RP-tree T' with n leaves and $\mathrm{TC}(T) = \mathrm{TC}(T') + n$. (The "$+n$" arises because each of the n leaves is one vertex further from the root in T than in T'.) If $\mathcal{A}(n)$ were false for T, it would also be false for T'. Thus we don't need to consider T at all if T has only one son. Thus it suffices to prove (8.1) when the root has degree 2. Let v_L and v_R be the two children of the root and let T_L and T_R be the trees rooted at v_L and v_R. Let k be the number of leaves in T_L. Then T_R has $n - k$ leaves. We have

$$\mathrm{TC}(T) \;=\; \Big(\mathrm{TC}(T_L) + k\Big) + \Big(\mathrm{TC}(T_R) + (n-k)\Big) \;\geq\; k\log_2 k + (n-k)\log_2(n-k) + n, \qquad 8.2$$

where the last part follows from $\mathcal{A}(k)$ and $\mathcal{A}(n-k)$ since $k < n$ and $n - k < n$. Clearly

$$k\log_2 k + (n-k)\log_2(n-k) \;\geq\; \min f(x), \qquad 8.3$$

where $f(x) = x\log_2 x + (n-x)\log_2(n-x)$ and the minimum is over all real x with $1 \leq x \leq n-1$.

This paragraph deals with the technicality of showing that the minimum of $f(x)$ is $n\log_2(n/2)$ and that it occurs at $x = n/2$. According to calculus we can find the minimum of $f(x)$ over an interval by looking at the values of $f(x)$ at the endpoints of the interval and when $f'(x) = 0$. The present endpoints are awkward (but they could be dealt with). It would be nicer to increase the interval to $0 \leq x \leq n$. To do this we must assign a value to $0\log_2 0$ so that $f(x)$ is continuous at 0. Thus we define $0\log_2 0$ to be

$$\begin{aligned}
\lim_{x\to 0^+} x\log_2 x \;&=\; \lim_{x\to 0^+} \frac{\log_2 x}{1/x} \\
&=\; \lim_{x\to 0^+} \frac{1/x\ln 2}{-1/x^2} \qquad \text{by l'Hôpital's Rule from calculus} \\
&=\; \lim_{x\to 0^+} \frac{-x}{\ln 2} = 0.
\end{aligned}$$

Hence $f(0) = f(n) = n\log_2 n$. Since

$$f'(x) \;=\; \log_2 x + \frac{x}{x\ln 2} - \log_2(n-x) - \frac{n-x}{(n-x)\ln 2} \;=\; \log_2\left(\frac{x}{n-x}\right)$$

and the logarithm is zero only at 1, it follows that $f'(x) = 0$ if and only if $\frac{x}{n-x} = 1$; that is, $x = n/2$. Since

$$f(n/2) = n\log_2(n/2) < n\log_2 n = f(0) = f(n),$$

the minimum of $f(x)$ occurs at $x = n/2$.

We have now shown that the right side of (8.3) is $n\log_2(n/2)$ and so, from (8.2),

$$\mathrm{TC}(n) \;\geq\; n\log_2(n/2) + n \;=\; n\log_2 n - n\log_2 2 + n \;=\; n\log_2 n.$$

This completes the induction step and hence the proof of the theorem. □

Exercises

8.1.1. In some data storage algorithms, information is stored at the leaves of a binary RP-tree and one reaches a leaf by answering a question at each vertex on the path from the root to a leaf, including at the leaf. (The question at the leaf is to allow for the possibility that the data may not be in the tree.) Suppose there are n items of data stored in such a tree. Show that the average number of questions required to recover stored data is at least $1 + \log_2 n$.

8.1.2. The average case result in Theorem 8.1 depends heavily on the fact that we expect to visit each leaf equally often. This may not be the case. For example, in some situations the list to be sorted is often nearly in order already. To see the importance of such frequency of visiting the leaves, consider a decision tree in which the relative frequency of visitation is 0.1, 0.2, 0.3, and 0.4. Find the tree for which $\mathrm{AC}(T)$ is a minimum.

8.1.3. Let T be a binary decision tree with n leaves. Suppose that $\mathrm{TC}(T)$ is the least it can be for a binary RP-tree with n leaves. Since the minimum in (8.3) occurred at $x = n/2$, one might expect the two principal subtrees of T to have a nearly equal number of leaves. But this is not the case. In this exercise, we explore what can be said.

(a) Prove that T is a full binary tree; i.e., there is no vertex with just one child.

(b) For any vertex in T, let $h(v)$ be the length of the path from v to the root of T. Let l_1 and l_2 be two leaves of T. Prove that $|h(l_1) - h(l_2)| \leq 1$.
Hint. Suppose $h(l_1) - h(l_2) \geq 2$. Let v be the parent of l_1 and consider what happens when l_2 and the subtree rooted at v are interchanged.

*(c) If $2^{m-1} < n \leq 2^m$, prove that the height of T is m. (The height of a tree is the maximum of $h(v)$ over all vertices v of T.)

*(d) If $2^{m-1} < n \leq 2^m$, prove that the maximum number of leaves that can be in a principal subtree of T is 2^{m-1}. Explain how to achieve this maximum.

8.1.4. In some sorts (e.g., a bucket sort described in the next section) a question may have more possible answers than just "yes" or "no." Suppose that each question has k possible answers. Show that the average number of questions required to sort n objects is at least $\log_k(n!)$. You may use the following fact without proof:
The minimum of $x_1 \log_k x_1 + \cdots + x_d \log_k x_d$ over all positive x_i that sum to n is obtained when all the x_i equal n/d and the value of the minimum is $n \log_k(n/d)$.

8.1.5. In some data storage and retrieval algorithms, a key and data are stored at each vertex of a binary RP-tree and is retrieved by means of a key. Suppose κ is the key whose data one wants. At each vertex v, one chooses the vertex (and stops) or one of the principal subtrees at v, depending on whether κ equals, exceeds, or is less than the key at v. Let $\mathrm{TC}^*(T)$ be the sum over all vertices v of T of the length of the path from v to the root.

(a) Show that a binary tree with no path to the root longer than n can store at most $2^{n+1} - 1$ keys.
Hint. Obtain a recursion for the number and use induction.

(b) Show that a tree T as in (a) storing the maximum number of possible keys has $\mathrm{TC}^*(T) = (n-1)2^{n+1} + 2$.

8.2 Software Sorts

A merge sort algorithm was studied in Example 7.13 (p. 211). You should review it now.

All reasonably fast software sorts use a *divide and conquer* method for attacking the problem. As you may recall, divide and conquer means splitting the problem into a few smaller problems which are easier either because they are smaller or because they are simpler. In problems where divide and conquer is most successful, it is often the case that the smaller problems are simply instances of the same type of problem and they are handled by applying the algorithm recursively. To give you a bit better idea of what divide and conquer means, here is how the algorithms we'll discuss use it. Some of this may not mean much to you until you've finished this section, so you may want to reread this list later. This is by no means an exhaustive list of the different types of software sorts.

- *Quicksort* and *merge sorts* split the data and spend most of their time sorting the separate pieces. Thus they divide and conquer by producing two smaller sorting problems which are handled in

a recursive manner.

Quicksort spends a little time dividing the data in such a way that recombining the two pieces after they are sorted is immediate. It divides the items into two collections so that all the items in the first collection should precede all the items in the second. The division is done "in place" by interchanging items that are in the wrong lists. Unless it is extremely unlucky, the two collections will have roughly the same number of elements. The two collections are then sorted separately.

Merge sorts reverse this: dividing is immediate and recombination takes a little time. In both cases, the "little time" required is proportional to the number of items being sorted because it requires a number of comparisons that is nearly equal to the number of items being sorted.

- An *insertion sort* builds up the sorted list by taking the items on the unsorted list one at a time and inserting them in a sorted list it is building. Divide and conquer can be used in the insertion process: To do a *binary insertion sort*, split the list into two nearly equal parts, decide which sublist should contain the new item, and iterate the process, using the sublist as the list.
- Suppose we are sorting a list of words (or numbers). *Bucket sort* focuses on one position in the words at a time. This is not usually a good divide and conquer approach because the task is not divided into just a few subproblems of roughly equal difficulty: On an n-long list with k characters per word, we focus in turn on each of the k positions. When n is large, k will be large, too.

  It is easy to get a time estimate for the algorithm. The amount of time it takes to process one character position for all n words is proportional to n. Thus, the time to sort is proportional to nk. How fast a bucket sort is depends on how large k is compared to n.
- *Heapsort* divides the sorting task into two simpler tasks.
  - First, the items are arranged in a structure, called a "heap," which is a rooted tree such that the smallest item in the tree is at the root and each of the sons of the root is also the root of a heap.
  - Second, the items are removed from the heap one by one, starting with the top and preserving the heap structure.

  Each of these two tasks requires about the same amount of time. Adding an item to the heap is done in a recursive manner, as is removing an item from the heap. The fact that a heap is defined recursively makes it easy to design recursive algorithms for manipulating heaps.

Binary Insertion Sort

Let $u_1, u_2 \ldots, u_n$ be the unsorted list. At the end of step t, the sorted list will be $s_1, s_2 \ldots, s_t$. At step t, an *insertion sort* determines where u_t belongs in the sorted list, opens up space and inserts it. For $t = 1$, this is trivial since the sorted list is empty. In general, we have a list $s_1, \ldots, s_{t-1}$. We must first find an index j such that $s_i \le u_t$ for $i \le j$ and $s_i \ge u_t$ for $i > j$, and then define a new sorted list by

$$\text{new } s_i \;=\; \begin{cases} \text{old } s_i & \text{if } 1 \le i < j; \\ u_t & \text{if } i = j; \\ \text{old } s_{i-1} & \text{if } j < i \le t. \end{cases}$$

How can we insure that a small number of comparisons is required for determining j? Simply searching the list from the start to find the place would, on average, take an amount of time proportional to k. A *binary insertion sort* uses divide and conquer to produce a much quicker insertion. It looks at the middle of the sorted list to decide which half should contain u_t and then iterates on that half until we are reduced to comparing u_t to a single item. This dividing of the list makes insertion in a k long list take one more comparison than insertion into a $k/2$ long list. Calculating the number of comparisons is left to Exercise 8.2.2.

Programming Note *In order to avoid moving a lot of items to insert u_t, insertion sorts are implemented by using a more complex data structure than a simple array. These data structures require more storage and somewhat more comparisons than binary insertion requires, but they require less movement of items. We will not discuss them. If you want further information, consult a good text on data structures and algorithms.*

Bucket Sort

As the name may suggest, a *bucket sort* is like throwing things into buckets. If we have a collection of buckets, we can put each item into the bucket where it belongs. If the buckets are in order and each bucket contains at most one item, then the items are sorted.

Since it is not practical to have this many buckets, a recursive method is used. This is better thought of in terms of piles instead of buckets. Suppose that you want to bucket sort the two digit numbers

22 31 12 23 13 33 11 21.

Here's how to do it.

1. Construct piles of numbers, one pile for each unit's digit, making sure that the order within a pile is the same as the order in the list. Here's the result.

   1: 31 11 21 2: 22 12 3: 23 13 33.

2. Make a list from the piles, preserving the order. Here's the result.

   31 11 21 22 12 23 13 33.

3. Repeat the first two steps using the new list and using the ten's digit instead of the unit's digit. Here are the results.

   1: 11 12 13 2: 21 22 23 3: 31 33

   gives

   11 12 13 21 22 23 31 33.

This method can be generalized to k digit numbers and k letter words. It is left as an exercise. In this case, each item is examined k times in order to place it in a bucket. Thus we place items in buckets kn times. If we think of digits as "letters," then a k digit number is simply a k letter word. Bucket sorting puts the words in lexicographic order. This sorting method can only be used on strings that are thought of as "words" made up of "letters" because we must look at items one "letter" at a time. The result is always a lexicographic ordering.

Suppose that we are sorting a large number n of k letter words where k is fairly small. In this case, kn, the number of times we have to place something in a bucket is less than the lower bound, $\log_2(n!)$, for the number of comparisons that we proved in the previous section. How can this be?

To begin with, deciding which bucket to place an item in is not a simple comparison unless there are only two buckets. If we want to convert that into a process that involves making decisions between pairs of items, we must do something like a binary insertion, comparing the item with the labels on the buckets. That will require about $\log_2 A$ comparisons, where A is the number of letters in the alphabet. Taking the number of comparisons into account, we obtain an estimate of $kn \log_2 A$ for the number of comparisons needed to bucket sort a list of n items.

It seems that we could simply keep k and A small and let n get large. This would still violate the lower bound of $\log_2(n!)$, which is about $n \log_2 n$. What has been ignored here is the fact that the lower bound was derived under the assumption that there can be $n!$ different orderings of a list. This can only happen if the list contains no duplicate words. Thus we must be able to create at least n

distinct words. Since there are A^k possible k letter words using an A letter alphabet, we must have $A^k \geq n$. Thus $k \log_2 A \geq \log_2 n$ and so $kn \log_2 A > n \log_2 n$, which agrees with our bound.

Programming Note *Hardware implementations of a bucket sort were in use for quite some time before inexpensive computers largely replaced them. They sorted "IBM cards," thin rectangular pieces of cardboard with holes punched in them by a "keypunch." The operator could set the sorting machine to drop the cards into bins according to the jth "letter" punched on the card. To sort cards by the word in columns i through j, the operator sorted on column j, restacked the cards, sorted on column $j-1$, restacked the cards, ..., sorted on column i and restacked the cards. In other words, a bucket sort was used. These machines were developed about a hundred years ago for dealing with United States census data. The company that manufactured them became IBM.*

In software bucket sorts, one usually avoids comparisons entirely by maintaining an A long table of pointers, one pointer for each bucket. The letter is then used as an index into the table to find the correct bucket.

Merge Sorts

We've discussed a simple merge sort already in Example 7.13 (p. 211). Recall that the list is divided arbitrarily into two pieces, each piece is sorted (by using the merge sort recursively) and, finally, the two sorted pieces are merged. The naïve method for merging two lists is to repeatedly move the smaller of the top items in the two lists to the end of the merged list we are constructing, but this cannot be implemented in a sorting network. The Batcher sort uses a more complex merging process that can be implemented in a sorting network. We'll study it in the next section.

Programming Note *For simplicity, assume that n is a power of 2. You can imagine the original list as consisting of n 1-long lists one after the other. These can be merged two at a time to produce $(n/2)$ 2-long lists one after the other. In general, we have $(n/2^k)$ 2^k-long lists which can be merged two at a time to produce $(n/2^{k+1})$ 2^{k+1}-long lists one after the other. If the data is on tape (see footnote page 227), one starts with two sets of 2^k-long lists and produces two sets of 2^{k+1}-long lists by merging the top two lists in each set and placing the merged lists alternately in the output sets. Since only sequential access is required, this simple merge sort is ideally suited to data on tape if four tape drives are available. There are variations of this idea which are faster because they require less tape movement.*

Quicksort

In Quicksort, an item is selected and the list is divided into two pieces: those items that should be before the selected item and those that should be after it. This is done in place so that one sublist precedes the other. If the two sublists are then sorted recursively using Quicksort, the entire original list will be sorted. About n comparisons are needed to divide the list.

How is the division accomplished? Here's one method. Memorize a list element, say x. Start a pointer at the left end of the list and move it rightward until something larger than x is encountered. Similarly, start a pointer moving leftward from the right end until something smaller than x is encountered. Switch the two items and start the pointers moving again. When the pointers reach the same item, everything to the left of the item is at most equal to x and everything to right of it is at least equal to x.

How long Quicksort takes depends on how evenly the list is divided. In the worst case, one sublist has one item and the remaining items are in the other. If this continues through each division, the

number of comparisons needed to sort the original list is about $n^2/2$. In the best case, the two sublists are as close to equal as possible. If this continues through each division, the number of comparisons needed to sort the original list is about $n \log_2 n$. The average number of sorts required is fairly close to $n \log_2 n$, but it's a bit tricky to prove it. We'll discuss it in Example 10.12 (p. 289).

Heapsort

We give a rough idea of the nature of Heapsort. A full discussion of Heapsort is beyond the scope of this text.

To explain Heapsort, we must define a *heap*. This data structure was described in a rough form above. Here is a complete, but terse, definition. A heap is a rooted binary tree with a bijection between the vertices and the items that are being sorted. The tree and bijection f satisfy the following.

- The smallest item in the image of f is associated with the root.
- The heights of the sons of the root differ by at most one.
- For each son v of the root, the subtree rooted at v and the bijection restricted to that subtree form a heap.

Thus the smallest case is a tree consisting of one vertex. This corresponds to a list with one element.

It turns out that it is fairly easy to add an item so that the heap structure is preserved and to remove the least item from the heap, in a way that preserves the heap structure. We will not discuss how a heap is implemented or how these operations are carried out. If you are interested in such details, see a text on data structures and algorithms.

Heapsort creates a heap from the unsorted list and then creates a sorted list from the heap by removing items one by one so that the heap structure is preserved. Thus the divide and conquer method in Heapsort involves the dividing of sorting into two phases: (a) creating the heap and (b) using the heap. Inserting or removing an item involves traversing a path between the root and a leaf. Since the greatest distance to a leaf in a tree with k nodes is about $\log_2 k$, the creation of the heap from an unsorted list takes about $\sum_{k=1}^{n} \log_2 k \approx n \log_2 n$ comparisons, as does the dismantling of the heap to form the sorted list. Thus Heapsort is a reasonably fast sort.

Note that a heap is an intermediate data structure which is quickly constructed from an unsorted list and which quickly leads to a sorted list. This observation is important. If we are in a situation where we need to continually add to a list and remove the smallest item, a heap is a good data structure to use. This is illustrated in the following example.

Programming Note *One way a heap can be implemented is with the classical form of a heap—a binary RP-tree in which*

- *each nonleaf node has a left son and, possibly, a right son;*
- *the longest and shortest distances from the root to leaves differ by at most one and*
- *an item is stored at each node in such a way that an item at a node precedes each of its sons in the sorted order.*

Note that the last condition implies that the smallest item is at the top of the heap. This tree can be stored as an array indexed from 1 to n. The sons of the item in position k are in positions $2k$ and $2k + 1$, if these numbers do not exceed n. With clever programming, the unsorted list, the heap and the sorted list can all occupy this space, so no extra storage is needed.

Exercises

8.2.1. (Binary Insertion Sort) Show the steps in a binary insertion sort of the list

$$9\ 15\ 6\ 12\ 3\ 7\ 11\ 5\ 14\ 1\ 10\ 4\ 2\ 13\ 16\ 8.$$

8.2.2. (Binary Insertion Sort) Show that the number of comparisons needed to find the location where u_t belongs is about $\log_2 t$. Use this to show that about $n \log_2 n$ comparisons are needed to sort an n long list by this method.
Hint. Using calculus, one can show that

$$\sum_{t=2}^{n} \log_2 t \ \approx \ \int_1^n x \log_2 x \, dx \ \approx \ n \log_2 n.$$

8.2.3. (Binary Insertion Sort) You are to construct the decision tree for the binary insertion sort. Label each leaf with the unsorted list written in one line form, as was done in Figure 8.1. If we are comparing an item with a sorted list of even length, there is no middle item. In this case, use the item just before the middle. This problem is a bit tricky. Unlike the tree in Figure 8.1, not all vertices at the same distance from the root involve the same comparison. As a help, you may want to label the vertices of the decision tree with the comparisons that are being made; for example, $i : j$ could mean that u_i is being compared to s_j.

(a) Construct the decision tree for the binary insertion sort on permutations of $\underline{2}$.

(b) Construct the decision tree for the binary insertion sort on permutations of $\underline{3}$.

8.2.4. (Bucket Sort) Show the steps in bucket sorting 33, 41, 23, 14, 21, 24.

8.2.5. (Bucket Sort)

(a) Carefully state an algorithm for bucket sorting words which have *exactly* k letters each.

(b) Carefully state an algorithm for bucket sorting words which have *at most* k letters each.

8.2.6. (Bucket Sort) Use induction on k to prove that the algorithms in the previous problem work.
Hint. By induction, after $k-1$ steps we have a list in which the items are in order if the first letter is ignored. What happens to this order within each of the piles we create when we look at the first letter?

8.2.7. (Merge Sort) Using the programming suggestion in the text, show on paper how merge sorting with 4 tapes works for the list

$$9\ 15\ 6\ 12\ 3\ 7\ 11\ 5\ 14\ 1\ 10\ 4\ 2\ 13\ 16\ 8.$$

(See the footnote on page 227 for an explanation of "tape.")

8.2.8. (Merge Sort) Suppose we have n items stored on a single tape and that we can sort k items at one time in memory. Explain how to reduce the number of times tapes need to be read for merge sort by first sorting sets of k items in memory.

8.2.9. (Quicksort) Prove that the number of comparisons required when the list is split as unevenly as possible is about $n^2/2$ as claimed. Prove that it is about $n \log_2 n$ when the splits are about as even as possible.

8.2.10. (Quicksort) Suppose that the each time a list of k items is divided in Quicksort the smaller piece contains rk items and the larger contains $(1-r)k$. Let $Q(n)$ be the number of comparisons needed to sort a list in this case.

(a) Show that $Q(n)$ is about $n + Q(rn) + Q((1-r)n)$.

(b) Present a reasonable argument (i.e., it need not be quite a proof) that $Q(n)$ is about $an \ln n$ where

$$1 + a(r \ln r + (1-r)\ln(1-r)) \;=\; 0.$$

(c) Verify that this gives the correct answer when the list is divided evenly each time ($r = 1/2$).

(d) It can be shown that $r = 1/3$ is a reasonable approximation for average behavior. What is $Q(n)$ in this case?

8.3 Sorting Networks

In sorting, items are compared and actions taken as a result of the comparison. Since items must be rearranged, the simplest kind of action one might visualize is to either do nothing or interchange the two items that were compared. We can imagine a hardware device for doing this, which we will call a *comparator*: It has two inputs, say x_1 and x_2, and two outputs, say y_1 and y_2, where y_1 is whichever one of x_1 and x_2 should appear earlier in the sorted list and y_2 is the other. A simple hardware device for sorting consists of a network of interconnected comparators. This is called a *sorting network*. For example, consider the following algorithm for sorting a permutation of $\underline{3}$.

1. If the entry in the second position exceeds the entry in the third position, switch them.
2. If the entry in the first position exceeds the entry in the second position, switch them.
3. If the entry in the second position exceeds the entry in the third position, switch them.

Figure 8.2 shows a sorting network to accomplish this. The data enters at the left end of the network and moves along the lines. The rearranged data emerges at the right end. A vertical connection between two lines represents a comparator—the two inputs emerge in sorted order.

Some of the questions we'd like to ask about sorting networks are

- How fast can sorting networks be?
- How many comparators are needed?
- How can we tell if a network sorts correctly?

8.3.1 Speed and Cost

In this section we'll focus on the first two questions that we raised above.

Sorting networks achieve speed by three methods: fast comparators, parallelism and pipelining. The first method is a subject for a course in hardware design, so we'll ignore it. Parallelism is built into any sorting network; we just haven't realized that in our discussion yet. Pipelining is a common design technique for speeding up computers.

Two things that will make a network cheaper to manufacture is decreasing the number of comparators it contains and increasing the regularity of the layout. Thus we can ask how many comparators are needed and if they can be arranged in a regular pattern.

Figure 8.2 Left: A sorting network with inputs a_i and outputs z_i. Vertical lines are comparators Right: What the network does to the inputs 3,2,1.

Figure 8.3 Left: A sorting network for the *Bubble Sort*. Right: The same network, but faster.

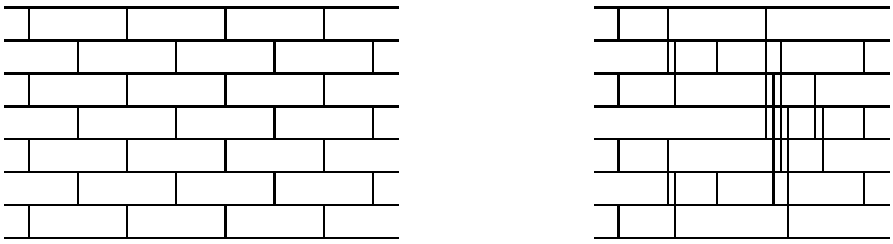


Figure 8.4 Left: A Brick Wall network for eight items. Right: The "Batcher" network for eight items is faster.

Parallelism

All the algorithms we've discussed so far have been carried out sequentially; that is, one thing is done at a time. This may not be realistic for algorithms on a supercomputer. It is certainly not realistic for sorting networks because parallelism is implicit in their design. Compare the two networks in Figure 8.3. They both perform the same sorting, but, as is evident from the second picture, some comparators can work at the same time.

In the expanded form on the left side, it is easy to see that the network sorts correctly. The leftmost set of comparators rising to the right finds the first item; the next set, the second item; the next, the third. The rightmost comparator puts the last two items in order. This idea can be extended to obtain a network that will sort $n > 1$ items in $2n - 3$ "time units," where a time unit is the length of time it takes a comparator to operate. Can we improve on this?

A natural idea is to fill in as many comparators as possible, thereby obtaining a *brick wall* type of pattern as shown in Figure 8.4. How long does the brick wall have to be? It turns out that, for n items, it must have length n. Can we do better than a brick wall if the comparators connect two lines which are not adjacent? Yes, we can. Figure 8.4 shows a "Batcher sort," which is faster than the brick wall. We'll explain Batcher sorts later. A vertical line spanning several horizontal lines represents a comparator connecting the topmost with the bottommost. To avoid overlapping lines in diagrams, comparators that are separated by a small horizontal distances in a diagram are understood to all be operating at the same time. The brick wall in Figure 8.4 takes 8 time units and the Batcher sort takes 6.

How Fast Can a Network Be?

Since a network must have at least $\log_2(n!)$ comparators and since at most $n/2$ comparators can operate at the same time, a network must take at least $\log_2(n!)/(n/2) \approx 2\log_2 n$ time units. It is known that for some C and for all large n, there exist networks that take no more than $C\log_2 n$ time units. This is too complicated for us to pursue here.

Pipelining is an important method for speeding up a network if many sets of items must be sorted. Suppose that we have a delay unit that delays an item by the length of time it takes a comparator to operate. Insert delay units in the network where there are no comparators so that all the items move through the network together. Once a set of items has passed a position, a new set can enter. Thus we can feed a new set of items into the network each time unit. The first set will emerge from the end some time later and then each successive set will take just one additional time unit to emerge. For example, a brick wall network can sort one set of 1,000 items in 1,000 time units, but 50 sets take only 1,049 time units instead of $1,000 \times 50$. This technique is known as pipelining because the network is thought of as a pipeline through which things are moving.

Pipelining is used extensively in computer design. It is obvious in the "vector processing" hardware of supercomputers, but it appears in some form in many central processing units. For example, the INTEL-8088 microprocessor can roughly be thought of as a two step pipeline: (i) the bus interface unit fetches instructions and (ii) the execution unit executes them. (It's actually a bit more complicated since the bus interface unit handles all memory read/write.)

How Cheap Can a Network Be?

Our previous suggestion for using a brick wall network to sort 1,000 items could be expensive—it requires 500,000 comparators. This number could be reduced by using a more efficient network; however, we'd probably have to sacrifice our nice regular pattern. There's another way we can achieve a dramatic saving if we are willing to abandon pipelining.

The brick wall network is simply the first two columns of comparators repeated about $n/2$ times. We could make a network that consists of just these two columns with the output feeding back in as input. Start the network by feeding in the desired values. After about n time units the sorted items will simply be circulating around in the network and can be read whenever we wish.

If we insist on pipelining, how many comparators are needed? From the exercises in the next section, you will see that a Batcher sorting network requires considerably less time and considerably less comparators than a brick wall network; however, it is not best possible. Unlike software sorting, there is a large gap between theory and practice in sorting nets: Theory provides a lower bound on the number of comparators that are required and specific networks provide an upper bound. There is a large gap between these two numbers. In contrast, the upper and lower bounds for pairwise comparisons in software sorts differ from $n\ln n$ by constant factors of reasonable size.

Whether or not we keep our pipelining capabilities, we face a variety of design tradeoffs in designing a VLSI chip to implement a sorting network. Among the issues are the number of comparators, the distance between the lines a comparator must connect, regularity of the design and delay problems. They are beyond the scope of our text.

Exercises

8.3.1. Find the fastest network you can for sorting 3 things and prove that there is no faster network.

8.3.2. Find the fastest network you can for sorting 4 things. If you've read the next section, prove that your network sorts correctly.

8.3.3. Find the fastest network you can for sorting 5 things. If you've read the next section, prove that your network sorts correctly.

8.3.4. Suppose we have a network for sorting n items and we wish to sort less than n. How can we use the network to do this?

8.3.5. In this exercise, comparators are only hooked to adjacent lines as in the brick wall. Find a network that will sort n inputs and that has as few comparators as you can manage.
Hint. Look at some small networks (e.g., $n = 3$ and $n = 4$) and try to find a simple pattern to generalize. If you read ahead a bit before working on this problem, you will encounter Theorem 8.3, which will help you prove that your network sorts.

8.3.6. Here's an idea for an even cheaper sorting network using the brick wall.

(a) Construct a two time unit network that consists of the first two time units of a brick wall. Feed the output of the network back as input. After some time $f(n)$, the output will be sorted regardless of the original input. Find the minimum value of $f(n)$ and explain why this behaves like a brick wall. (Note, that using the network k times means that $2k$ time units have elapsed.)

(b) What can be done if we have such an arrangement for sorting n items and we wish to sort less than n?

(c) When n is even, we can get by with even less comparators. Construct a network that compares the items $2i - 1$ and $2i$ for $1 \leq i \leq n/2$. Call the inputs $x_1, \ldots, x_n$ and the outputs $y_1, \ldots, y_n$. Feed the output of the network back in as follows:

$$\text{new } x_k \;=\; \begin{cases} \text{old } y_n & \text{if } k = 1; \\ \text{old } y_{k-1} & \text{otherwise.} \end{cases}$$

At time $2k$ disable the comparator between lines $2k - 1$ and $2k$. Show that this sorts after some number of steps. How many steps are needed in general?

8.3.2 Proving That a Network Sorts

Unlike many software sorts, it is frequently difficult to prove that a network actually sorts all inputs correctly. There is no panacea for this problem. The following two theorems are sometimes helpful.

Theorem 8.2 Zero-One Principle *If a network correctly sorts all inputs of zeroes and ones, then it correctly sorts all inputs.*

Theorem 8.3 Adjacent Comparisons *If the comparators in a network only connect adjacent lines and if the network correctly sorts the reversed sequence $n, \ldots, 2, 1$, then it correctly sorts all inputs.*

We will prove the Zero-One Principle shortly. The proof of the other theorem is more complicated and will not be given.

Since the Adjacent Comparisons Theorem requires that only one input be checked, it is quite useful. Unfortunately, the comparators must connect adjacent lines. To see that this is needed, consider a three input network in which the top and bottom lines are connected by a comparator and there are no other comparators. It correctly sorts the inputs 1,2,3 and 3,2,1, but it does not sort any other permutations of 1,2,3 correctly.

The Zero-One Principle may seem somewhat useless because it still requires that many inputs be considered. We will see that it is quite useful for proving that a Batcher sort works.

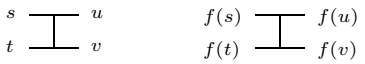


Figure 8.5 Left: A single comparator with arbitrary inputs. Right: Applying a nondecreasing function.

Proof: We now prove the Zero-One Principle. If the network fails to sort some sequence, we will show how to construct a sequence of zeroes and ones that it fails to sort.

The idea behind our proof is the following simple observation: Suppose that f is a nondecreasing function, then a comparator treats $f(s)$ and $f(t)$ the same as it does s and t. This is illustrated in Figure 8.5. It is easy to show by considering the three cases $s < t$, $s = t$ and $s > t$.

Suppose a network has N comparators, has inputs $x_1, \ldots, x_n$ and outputs $y_1, \ldots, y_n$. Let f be a nondecreasing function. We will use induction on N to prove that if $f(x_1), \ldots, f(x_n)$ are fed into the network, then $f(y_1), \ldots, f(y_n)$ emerge at the other end.

If $N = 0$, the result is obviously true since there are no comparators present.

Now suppose that $N > 0$ and that the result is true for $N - 1$. We will prove it for N. Focus on one of the comparators that is used in the last time unit. Let network $\mathcal{N}_1$ be the original network with this comparator removed and let network $\mathcal{N}_2$ be the original network with all but this comparator removed. Let $z_1, \ldots, z_n$ be the output of $\mathcal{N}_1$ and use it as the input to $\mathcal{N}_2$. Clearly the output of $\mathcal{N}_2$ is $y_1, \ldots, y_n$.

Let $f(x_1), \ldots, f(x_n)$ be input to the network and let $u_1, \ldots, u_n$ be the output. We can break this into two pieces:

- Let $f(x_1), \ldots, f(x_n)$ be the input to $\mathcal{N}_1$.
  By the induction hypothesis, the output is $f(z_1), \ldots, f(z_n)$.
- Let $f(z_1), \ldots, f(z_n)$ be the input to $\mathcal{N}_2$.
  The output is then $u_1, \ldots, u_n$.

We must prove that $u_1, \ldots, u_n$ is $f(y_1), \ldots, f(y_n)$.

Recall that $\mathcal{N}_2$ consists of a single comparator. Let the input positions i and j go into the comparator. If $k \neq i, j$, then that input position does not go through the comparator and so $f(z_k)$ is treated the same as z_k. Our observation for a single comparator applies to the input positions for z_i and z_j since they feed into the comparator. This proves that $u_1, \ldots, u_n$ equals $f(y_1), \ldots, f(y_n)$ and so proves our claim about nondecreasing functions.

Now suppose that the network fails to sort $x_1, \ldots, x_n$. We will show how to construct a nondecreasing 0-1 valued function f such that the network fails to sort $f(x_1), \ldots, f(x_n)$. Since the sort fails, we must have $y_i > y_{i+1}$ for some $i < n$. Define

$$f(t) = \begin{cases} 0, & \text{if } t < y_i; \\ 1, & \text{if } t \geq y_i. \end{cases}$$

Since $y_{i+1} < y_i$, we have $f(y_{i+1}) = 0$. Because $f(y_i) = 1$, the network fails to sort $f(x_1), \ldots, f(x_n)$. □

The Batcher Sort

The Batcher sort is a merge sort that is suitable for implementation using a sorting network. Like all merge sorts, it is defined recursively. Let $k = \lceil n/2 \rceil$ be the result of rounding $n/2$ up; for example, $\lceil 3.5 \rceil = 4$. For a Batcher sort, the first k items are sorted and the last $n - k$ are sorted. Then the two sorted sublists are merged.

To write pseudocode for the Batcher sort, let `Comparator`(x, y) replace x with the smaller of x and y and replace y with the larger. The Batcher sort for array $x_1, \ldots, x_n$ is as follows.

```
BSORT(x_1, ..., x_n)
     If n = 1
          Return
     End if
     k = ⌈n/2⌉
     BSORT(x_1, ..., x_k)                    /* Do these in ... */
     BSORT(x_{k+1}, ..., x_n)                /* ... parallel. */
     BMERGE(x_1, ..., x_n)
     Return
End

BMERGE(x_1, ..., x_n)
     If n = 1
          Return
     End if
     If n = 2
          Comparator(x_1, x_2)
          Return
     End if
     BMERGE2(x_1, ..., x_k; x_{k+1}, ..., x_n)
     For i = 1, 2, ..., ⌈n/2⌉ - 1
          Comparator(x_{2i},x_{2i+1})        /* Do these in parallel. */
     End for
     Return
End

BMERGE2(x_1, ..., x_k; y_1, ..., y_j)
     BMERGE(x_1, x_3, x_5, ..., y_1, y_3, y_5, ...)   /* Do these in ... */
     BMERGE(x_2, x_4, x_6, ..., y_2, y_4, y_6, ...)   /* ... parallel. */
     Return
End
```

The first procedure is the usual form for a merge sort. The other two procedures are the Batcher merge. They could be combined into one procedure, but keeping track of subscripts gets a bit messy.

The Batcher sort for eight items is shown in Figure 8.6 with some parts labeled. The part labeled S_2 is four copies of a two item Batcher sort. The parts labeled C_n are the comparators in `BMERGE` on a list of n items. From S_2 through C_4 inclusive is two copies of a four item Batcher sort, one for the top four inputs and one for the bottom four. The parts labeled O and E are the odd and even indexed entries being treated by `BMERGE` in the call of `BMERGE2` with an eight long list.

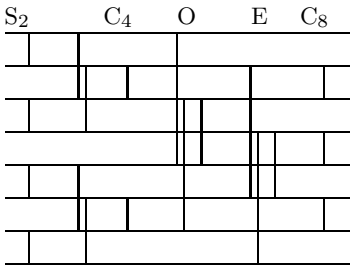


Figure 8.6 The Batcher network for eight items.

To prove that the Batcher sort works, it suffices to prove that the merge part works. Why is this? *Any* recursive sort with a merge will work: The sequence is split in two, each part is sorted and the two parts are merged.

A variation of the Zero-One Principle can be used to prove that the merge works. We need only prove the merge for all sequences of zeroes and ones for which both the first and second halves have been sorted. We remark that $j \leq k \leq j+1$ whenever `BMERGE2` is called. The idea of the proof is to use induction on n. A key observation is that the number of zeroes in the two sequences that `BMERGE2` passes to `BMERGE` are practically the same: The number of zeroes in the sequence made from the odd subscripted x's and y's less the number of zeroes in the other sequence is 0, 1 or 2. One can then consider the three possible cases separately. This is left as an exercise.

Exercises

8.3.7. Prove that if the comparators in a network only connect adjacent lines and if the network correctly sorts all sequences that consist of ones followed by zeroes, then it correctly sorts all inputs.

8.3.8. (Brick wall network correctness) This exercise is a preparation for the one to follow.

(a) Draw a diagram to show how the sequence 5,4,3,2,1 moves through the brick wall network. Show how to cut out all appearances of 5 and push the network together to obtain a diagram for 4,3,2,1.

(b) Draw diagrams to show how the two input sequences 1,1,0 and 1,0,0 move through the brick wall.

(c) Repeat the previous part for the sequences 1,1,1,0 and 1,1,0,0 and 1,0,0,0.

8.3.9. (Brick wall network correctness) Prove that the brick wall correctly sorts all inputs. You may find an idea for a proof in the previous exercise.

8.3.10. Draw networks for Batcher sorts for n up to 8.

8.3.11. (Batcher merge correctness) Fill in the details of the proof that `BMERGE` works when n is a power of 2; that is, if $x_1, \ldots, x_k$ are in order and $x_{k+1}, \ldots, x_n$ are sorted, then `BMERGE`$(x_1, \ldots, x_n)$ results in a sorted list.
Hint. In this case, division always comes out even and $k = j = n/2$.

8.3.12. (Batcher merge correctness) Fill in the details of the proof that `BMERGE` works for all n; that is, if $x_1, \ldots, x_k$ are in order and $x_{k+1}, \ldots, x_n$ are sorted, then `BMERGE`$(x_1, \ldots, x_n)$ results in a sorted list.

8.3.13. (Batcher network time) Let $S(N)$ be the number of time units a Batcher network takes to sort 2^N things and let $M(N)$ be the number of time units a Batcher network takes to execute `BMERGE` on 2^N things.

(a) Prove that $S(0) = 0$ and, for $N > 0$, $S(N) \leq S(N-1) + M(N)$.

(b) Prove that $M(0) = 0$ and, for $N > 0$, $M(N) \leq M(N-1) + 1$.

(c) Conclude that $S(N) \leq N(N+1)/2$.

(d) What can you say about the time it takes a Batcher network to sort n things when n is not a power of two?

Notes and References

Most books on combinatorial algorithms discuss software algorithms for sorting and a few discuss hardware algorithms; i.e., sorting nets. You may wish to look at the references at the end of Chapter 6. The classic reference is Knuth [4] where you can find more details on the various software sorts that we've discussed. See also pp. 47–75 of Williamson [5].

"Expander graphs" are the basis of sorting networks that require only $\Theta(n \ln n)$ comparators. At present, they have not yet led to any practical networks. For a discussion of this topic see Chapter 5 of the book by Gibbons and Rytter [3]. It is possible to construct fairly cheap networks based on a modified Batcher sort. Like the brick wall, these networks consist of a pattern repeated many times. The brick wall is a 2-long pattern repeated $n/2$ times. The modified Batcher network is a $(\log_2 n)$-long pattern repeated $(\log_2 n)$ times. See [2] for details. See also [1]. A proof of Theorem 8.3 can be found on p. 63 of [5].

1. Edward A. Bender and S. Gill Williamson, Periodic sorting using minimum delay, recursively constructed sorting networks, *Electronic J. Combinatorics* **5** (1998), R5.
2. E. Rodney Canfield and S. Gill Williamson, A sequential sorting network analogous to the Batcher merge, *Linear and Multilinear Algebra* **29** (1991), 42–51.
3. Alan Gibbons and Wojciech Rytter, *Efficient Parallel Algorithms*, Cambridge Univ. Press (1988).
4. Donald E. Knuth, *The Art of Computer Programming. Vol. 3 Sorting and Searching*, 2nd ed., Addison-Wesley (1998).
5. S. Gill Williamson, *Combinatorics for Computer Science*, Dover (2002).

CHAPTER 9

Rooted Plane Trees

Introduction

The most important recursive definition in computer science may be the definition of an *rooted plane tree* (RP-tree) given in Section 5.4. For convenience and emphasis, here it is again.

Definition 9.1 Rooted plane trees *An* **RP-tree** *consists of a set of vertices each of which has a (possibly empty) linearly ordered list of vertices associated with it called its* **sons** *or* **children**. *Exactly one of the vertices of the tree is called the* **root**. *Among all such possibilities, only those produced in the following manner are RP-trees.*

- *A single vertex with no sons is an RP-tree. That vertex is the root.*
- *If $T_1, \ldots, T_k$ is an ordered list of RP-trees with roots $r_1, \ldots, r_k$ and no vertices in common, then an RP-tree T can be constructed by choosing an unused vertex r to be the root, letting its ith child be r_i and forgetting that $r_1, \ldots, r_k$ were called roots.*

In Example 7.9 (p. 206) we sketched a proof that this definition agrees with the one given in Definition 5.12. Figure 9.1 illustrates the last stage in building an RP-tree by using our new recursive definition. In that case $k = 3$.

We need a little more terminology. The RP-trees $T_1, \ldots, T_k$ in the definition are called the *principal subtrees* of T. A vertex with no sons is called a *leaf*.

The fact that a general RP-tree can be defined recursively opens up the possibility of recursive algorithms for manipulating general classes of RP-trees. Of course, we are frequently interested in special classes of RP-trees. Those special classes which can be defined recursively are often the most powerful and elegant. Here are three such classes.

- In Example 7.14 (p. 213) we saw how a local description could be associated with a recursive algorithm. In Example 7.16 (p. 214) a local description was expanded into a tree for the Tower of Hanoi procedure. These local descriptions are simply recursive descriptions of RP-trees that describe the algorithms. A leaf is the "output" of the algorithm. In these two examples, a leaf is either a permutation or the movement of a single washer, respectively. In other words:

A recursive algorithm
contains
a local description
which is
a recursive definition of some RP-trees.

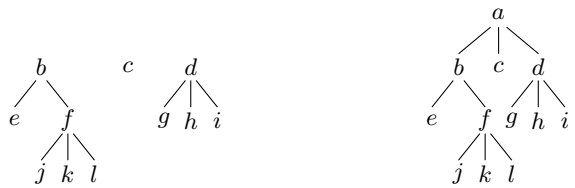


Figure 9.1 The last stage in recursively building an RP-tree. Left: The trees T_1, T_2 and T_3 with roots b, c and d. Right: The new tree T with root a.

In Section 9.1, we'll look at some recursive algorithms for traversing RP-trees.

- Compilers are an important aspect of computer science. Associated with a statement in a language is a "parse tree." This is an RP-tree in which the leaves are the parts of the language that you actually see and the other vertices are grammatical terms. "Context free" grammars are recursively defined and lead to recursively defined parse trees. In Section 9.2 we'll briefly look at some simple parse trees.
- Those RP-trees in which each vertex has either zero or two sons are called *full binary RP-trees*. We'll study them in Section 9.3, with emphasis on ranking and unranking them. (Ranking and unranking were studied in Section 3.2.) Since the trees are defined recursively, so is their rank function.

Except for a reference in Example 9.12 (p. 263), *the sections of this chapter can be read independently of one another.*

9.1 Traversing Trees

A *tree traversal algorithm* is a systematic method for visiting all the vertices in an RP-tree. We've already seen a nonrecursive traversal algorithm in Theorem 3.5 (p. 85). As we shall soon see a recursive description is much simpler. It is essentially a local description.

Traversal algorithms fall into two categories called "breadth first" and "depth first," with depth first being the more common type. After explaining the categories, we'll focus on depth first algorithms.

The left side of Figure 9.2 shows an RP-tree. Consider the right side of Figure 9.2. There we see the same RP-tree. Take your pencil and, starting at the root a, follow the arrows in such a way that you visit the vertices in the order

$$a\ b\ e\ b\ f\ j\ f\ k\ f\ l\ f\ b\ a\ c\ a\ d\ g\ d\ h\ d\ i\ d\ a. \tag{9.1}$$

This manner of traversing the tree diagram, which extends in an obvious manner to any RP-tree T, is called a *depth-first traversal* of the ordered rooted tree T. The sequence of vertices (9.1) associated with the depth-first traversal of the RP-tree T will be called the *depth-first vertex sequence* of T and will be denoted by $\mathrm{DFV}(T)$. If you do a depth-first traversal of an RP-tree T and list the edges encountered (list an edge each time your pencil passes its midpoint in the diagram), you obtain the *depth-first edge sequence* of T, denoted by $\mathrm{DFE}(T)$. In Figure 9.2, the sequence $\mathrm{DFE}(T)$ is

$$\begin{array}{cccccccccc} \{a,b\} & \{b,e\} & \{b,e\} & \{b,f\} & \{f,j\} & \{f,j\} & \{f,k\} & \{f,k\} & \{f,l\} & \{f,l\} \\ \{b,f\} & \{a,b\} & \{a,c\} & \{a,c\} & \{a,d\} & \{d,g\} & \{d,g\} & \{d,h\} & \{d,h\} & \{d,i\} \\ \{d,i\} & \{a,d\}. & & & & & & & & \end{array}$$

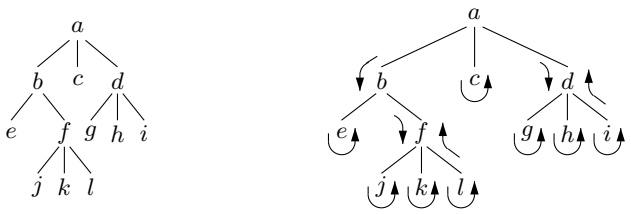


Figure 9.2 Left: An RP-tree with root a. Right: Arrows show depth first traversal of the tree.

The other important linear order associated with RP-trees is called *breadth-first order*. This order is obtained, in the case of Figure 9.2, by reading the vertices or edges level by level, starting with the root. In the case of vertices, we obtain the *breadth-first vertex sequence* (BFV(T)). In Figure 9.2, BFV(T) = $abcdefghijkl$. Similarly, we can define the *breadth-first edge sequence* (BFE(T)).

Although we have defined these orders for trees, the ideas can be extended to other graphs. For example, one can use a breadth first search to find the shortest (least number of choices) route out of a maze: Construct a decision tree in which each vertex corresponds to an intersection in the maze. (More than one vertex may correspond to the same intersection.) A vertex corresponding to an intersection already encountered in the breadth first search has no sons. The decisions at an intersection not previously encountered are all possibilities of the form "follow a passage to the next intersection."

Example 9.1 Data structures for tree traversals Depth-first and breadth-first traversals have data structures naturally associated with their computer implementations.

BFV(T) can be implemented by using a queue. A *queue* is a list from which items are removed at the opposite end from which they are added (first in, first out). Checkout lines at markets are queues. The root of the tree is listed and placed on the queue. As long as the queue is not empty, remove the next vertex from the queue and place its sons on the queue. You should be able to modify this to give BFE(T)

DFV(T) can be implemented by using a stack. A *stack* is a list from which items are removed at the same end to which they are added (last in, first out). They are used in computer programming to implement recursive code. (See Example 7.17 (p. 216).) For DFV, the root of the tree is listed and placed on the stack. As long as the stack is not empty, remove the vertex that is on the top of the stack from the stack and place its sons, in order, on the stack so that leftmost son is on the top of the stack.

When a vertex is added to or removed from the data structure, you may want to take such action; otherwise, you have will have traversed the tree without accomplishing anything.

Depth First Traversals

In a tree traversal, we often want to process either the vertices or edges and do so either the first or last time we encounter them. If you process something only the first time it is encountered, this is a *preorder traversal*; if you list it only the last time it is encountered, this is a *postorder traversal*; This leads to four concepts:

PREV(T)	*preorder vertex sequence;*
POSTV(T)	*postorder vertex sequence;*
PREE(T)	*preorder edge sequence;*
POSTE(T)	*postorder edge sequence.*

Here's the promised recursive algorithm for depth first traversal of a tree. The sequences PREV, POSTV, PREE and POSTE are initialized to empty. They are "global variables," so all levels of the recursive call are working with the same four sequences.

```
Procedure DFT(T)
     Let r be the root of T
     Append vertex r to PREV              /* PREV */
     Let k be the number of principal subtrees of T
     /* By convention, the For loop is skipped if k = 0. */
     For i = 1, 2, ..., k
          Let T_i be the ith principal subtree of T
          Let r_i be the root of T_i
          Append edge {r, r_i} to PREE    /* PREE */
          DFT(T_i)
          Append edge {r, r_i} to POSTE   /* POSTE */
          Append vertex r to POSTV        /* POSTV */
     End for
     Return
End
```

For example, in Figure 9.2, PREV(T) = $abefjklcdghi$ and POSTV(T) = $ejklfbcghida$. Our pseudocode is easily modified to do these traversals: Simply cross out those "`List`" lines whose comments refer to traversals you don't want.

When a tree is being traversed, the programmer normally does not want a list of the vertices or edges. Instead, he wants to take some sort of action. Thus "`Append vertex` v ..." and "`Append edge` $\{u, v\}$ ..." would probably be replaced by something like "`DoVERTEX`(v)" and "`DoEDGE`(u, v)," respectively.

Example 9.2 Reconstructing trees

Does a sequence like PREV(T) have enough information to reconstruct the tree T from the sequence? Of course, one might replace PREV with other possibilities such as POSTV.

The answer is "no". One way to show this is by finding two trees T and U with PREV(T) = PREV(U) — or using whatever other possibility one wants in place of PREV. The trouble with this approach is that we need a new example for each case. We'll use the Pigeonhole Principle (p. 55) to give a proof that is easily adapted to other situations. Given a set V of n labels for the vertices, there are $n!$ possible sequences for PREV since each such sequence must be a permutation of V. Let B be the set of these permutations and let A be the set of RP-trees with vertex set V. Then PREV : $A \to B$ and we want to show that two elements of A map to the same permutation. By the Pigeonhole Principle, it suffices to show that $|A| > |B|$. We already know that $|B| = n!$ There are n^{n-2} trees by Example 5.10 (p. 143). Since many RP-trees may give the same tree (root and order information is removed), we have $|A| \geq n^{n-2}$. We need to show that $n! < n^{n-2}$ for some value of n. This can be done by finding such an n or by using Stirling's Formula, Theorem 1.5 (p. 12). We leave this to you. □

Example 9.3 Graph traversal and spanning trees One can do depth first traversal of a graph to construct a lineal spanning tree a concept defined in Definition 6.2 (p. 153). The following algorithm finds a lineal spanning tree with root r for a connected simple graph $G = (V, E)$. The graph is a "global variable," so changes to it affect all the levels of the recursive calls. When a vertex is removed from G, so are the incident edges. The comments refer to the proof of Theorem 6.2 (p. 153). We leave it to you to prove that the algorithm does follow the proof as claimed in the comments.

```
/* Generate a lineal spanning tree of G rooted at r. */
LST(r)
     If (no edges contain r)
          Remove r from G
          Return the 1 vertex tree with root r
     End if
     /* f = {r,s} is as in the proof. */
     Choose {r,s} ∈ E
     Remove r from G, saving it and its incident edges
     /* S corresponds to T(A) in the proof. */
     S = LST(s)
     Restore r and the saved edges whose ends are still in G
     /* R corresponds to T(B) in the proof. */
     R = LST(r)
     Join S to R by an edge {r,s} to obtain a new tree T
     Remove r from G
     Return T
End   □
```

Example 9.4 Counting RP-trees When doing a depth first traversal of an unlabeled RP-tree, imagine listing the direction of each step: a for away from the root and t for toward the root. What can we see in this sequence of a's and t's?

Each edge contributes one a and one t since it is traversed in each direction once. If the tree has n edges, we get a $2n$-long sequence containing n copies of a and n copies of t, say $s_1, \ldots, s_{2n}$. If $s_1, \ldots, s_k$ contains d_k more a's than t's, we will be a distance d_k from the root after k steps because we've taken d more steps away from the root than toward it. In particular $d_k \geq 0$ for all k.

Thus a tree with n edges determines a unique sequence of n a's and n t's in which each initial part of the sequence contains at least as many a's as t's. Furthermore, you should be able to see how to reconstruct the tree if you are given such a sequence. Thus there is a bijection between the trees and the sequences. It follows from the first paragraph of Example 1.13 (p. 15) that the number of n-edge unlabeled RP-trees is C_n, the Catalan number.

Since a tree with n vertices has $n - 1$ edges, it follows that the number of n-vertex unlabeled RP-trees is C_{n-1}. By Exercise 9.3.12 (p. 266), it follows that the number of unlabeled binary RP-trees with n leaves is C_{n-1}. Thus the solution to Exercise 9.3.13 provides a formula for the Catalan numbers. □

Exercises

9.1.1. Write pseudocode for recursive algorithms for PREV(T), PREE(T) and POSTE(T).

9.1.2. In the case of decision trees we are only interested in visiting the leaves of the tree. Call the resulting sequence DFL(T) Write pseudocode for a recursive algorithm for DFL(T). What is the connection with the traversal algorithm in Theorem 3.5 (p. 85)?

9.1.3. Write pseudocode to implement breadth first traversal using a queue. Call the operations for adding to the queue and removing from the queue `INQUEUE` and `OUTQUEUE`, respectively.

9.1.4. Write pseudocode to implement PREV(T) using a stack. Call the operations for adding to the stack and removing from the stack `PUSH` and `POP`, respectively.

9.1.5. Construct a careful proof that the algorithm in Example 9.3 does indeed construct a lineal spanning tree.

9.1.6. In contrast to Example 9.2, if PREV(T) = PREV(U) *and* POSTV(T) = POSTV(U), then $T = U$. Another way to state this is:

$$\text{Given PREV}(T) \text{ and POSTV}(T), \text{ we can reconstruct } T. \tag{9.2}$$

The goal of this exercise is to prove (9.2) by induction on the number of vertices.

Let $n = |T|$, the number of vertices of T. Let v be the root of T. Let PREV(T) = $a_1, \ldots, a_n$ and POSTV(T) = $z_1, \ldots, z_n$.

If $n > 1$, we may suppose that T consists of the RP-trees $T_1, \ldots, T_k$ joined to the root v. Let U be the RP-tree with root v joined to the trees $T_2, \ldots, T_k$. (If $k = 1$, U is just the single vertex v.)

(a) Prove (9.2) is true when $n = 1$.

(b) Prove that a_2 is the root of T_1

(c) Suppose $z_t = a_2$. (This must be true for some t since PREV and POSTV are both permutations of the vertex set.) Prove that POSTV(T_1) = $z_1, \ldots, z_t$.

(d) With t as above, prove that PREV(T_1) = $a_2, \ldots, a_{t+1}$.
Hint. How many vertices are there in T_1?

(e) Prove that PREV(U) = $v, a_{t+2}, \ldots, a_n$ and POSTV(U) = $z_{t+1}, \ldots, z_n$.

(f) Complete the proof.

9.1.7. In Example 9.2, we proved that a tree cannot be reconstructed from the sequence PREV. The proof also works for POSTV.

(a) Find two different RP-trees T and U with PREV(T) = PREV(U).

(b) Find two different RP-trees T and U with POSTV(T) = POSTV(U).

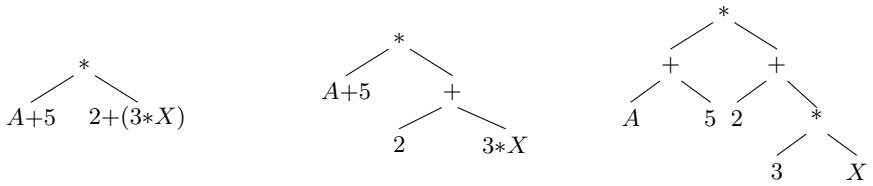
9.1.8. Use Exercise 9.1.6 to write pseudocode for a recursive procedure to reconstruct a tree from PREV and POSTV.

9.1.9. If T is an unlabeled RP-tree, define a sequence $D(T)$ of ± 1's as follows. Perform a depth first traversal of the tree. Whenever an edge is followed from father to son, list a $+1$. Whenever an edge is followed from son to father, list a -1.

(a) Let $T_1, \ldots, T_m$ be unlabeled RP-trees and let T be the tree formed by joining the roots of each of the T_i's to a new root to form a new unlabeled RP-tree. Express $D(T)$ in terms of $D(T_1)$, ..., $D(T_m)$.

(b) IF $D(T) = s_1, \ldots, s_n$, show that n is twice the number of edges of T and show that $\sum_{i=1}^{k} s_i \geq 0$ for all k with equality when $k = n$.

*(c) Let $\vec{s} = s_1, \ldots, s_n$ be a sequence of ± 1's. Show that $\vec{s}$ comes from at most one tree and that it comes from a tree if and only if $\sum_{i=1}^{k} s_i \geq 0$ for all k, with equality when $k = n$.



(a) The first step. (b) An intermediate step. (c) The final result.

Figure 9.3 Interpreting the expression $(A + 5) * (2 + (3 * X))$.

9.2 Grammars and RP-Trees

Languages are important in computer science. Compilers convert programming languages into machine code. Automatic translation projects are entangled by the intricacies of natural languages. Researchers in artificial intelligence are interested in how people manage to understand language. We look briefly at a simple, small part of all this: "context-free grammars" and "parse trees." To provide some background material, we'll look at arithmetic expressions and at simple sentences.

Unfortunately, we'll be introducing quit a bit of terminology. Fortunately, you won't need most of it for later sections, so, if you forget it, you can simply look up what you need in the index at the time it is needed.

Example 9.5 Arithmetic expressions Let's consider the meaning of the expression $(A + 5) * (2 + (3 * X))$.

It means $A + 5$ times $2 + (3 * X)$, which we can represent by the RP-tree in Figure 9.3(a). We can then interpret the subexpressions and replace them by their interpretations and so on until we obtain Figure 9.3(c). We can represent this recursive procedure as follows, where "*exp*" is short for "*expression*".

```
INTERPRET(exp)
     If (exp has no operation)
          Return the RP-tree with root exp
               and no other vertices.
     End if
     Let exp = (exp_l op exp_r).
     Return the RP-tree with root op,
          left principal subtree INTERPRET(exp_l) and
          right principal subtree INTERPRET(exp_r).
End
```

Now suppose we wish to evaluate the expression, with a procedure we'll call `EVALUATE`. One way of doing that would be to modify `INTERPRET` slightly so that it returns values instead of trees. A less obvious method is to traverse the tree generated by `INTERPRET` in POSTV order. Each leaf is replaced by its value and each nonleaf is replaced by the value of performing the operation it indicates on the values of its two sons.

To illustrate this, POSTV of Figure 9.3(c) is $A\ 5\ +\ 2\ 3\ X\ *\ +\ *$. Thus we replace the leaves labeled A and 5 with their values, then the leftmost vertex labeled + with the value of $A + 5$, and so on. POSTV of a tree associated with calculations is called *postorder notation* or *reverse Polish notation*. It is used in some computer languages such as Forth and PostScript® and in some handheld calculators. □

Example 9.6 Very simple English Languages are very complicated creations; however, we can describe a rather simple form of an English sentence as follows:

(a) One type *sentence* consists of the three parts *noun-phrase*, *verb*, *noun-phrase* and a period in that order.

(b) A *noun-phrase* is either a *noun* or an *adjective* followed by a *noun-phrase*.

(c) Lists of acceptable words to use in place of *verb*, *noun* and *adjective* are available.

This description is a gross oversimplification. It could lead to such sentences as "**big brown small houses sees green green boys.**" The disagreement between subject and verb (plural versus singular) could be fixed rather easily, but the nonsense nature of the sentence is not so easily fixed. If we agree to distinguish between grammatical correctness and content, we can avoid this problem. (As we shall see in a little while, this is not merely a way of defining our difficulty out of existence.) □

Let's rephrase our rules from the last example in more concise form. Here's one way to do that.

(a) *sentence* → *noun-phrase verb noun-phrase* **.**

(b) *noun-phrase* → *adjective noun-phrase* | *noun*

(c) *adjective* → **big** | **small** | **green** | ⋯

noun → **houses** | **boys** | ⋯

verb → **sees** | ⋯

These rules along with the fact that we are interested in sentences are what is called a "context-free grammar."

Definition 9.2 Context-free grammar *A* **context-free grammar** *consists of*

1. *a finite set S of* **nonterminals***;*
2. *a finite set T of* **terminals***;*
3. *a* **start symbol** $s_0 \in S$*;*
4. *a finite set of* **productions** *of the form* $s \to x_1 \dots x_n$ *where* $s \in S$ *and* $x_i \in S \cup T$*. We allow* $n = 0$*, in which case the production is "*$s \to$*."*

*We'll distinguish between terminal and nonterminal symbols by writing them in the fonts "***terminal***" and "nonterminal." If we don't know whether or not an element is terminal, we will use the nonterminal font. (This can happen in a statement like,* $x \in S \cup T$*.)*

In the previous example, the start symbol is *sentence* and the productions are given in (a)–(c).

The productions of the grammar give the structural rules for building the language associated with the grammar. This set of rules is called the *syntax* of the language. The grammar is called context-free because the productions do not depend on the context (surroundings) in which the nonterminal symbol appears. Natural languages are not context-free, but many computer languages are nearly context-free.

Any string of symbols that can be obtained from the start symbol (*sentence* above) by repeated substitutions using the productions is called a sentential form.

Definition 9.3 The language of a grammar *A sentential form consisting only of terminal symbols is a* **sentence***. The set of sentences associated with a grammar G is called the* **language** *of the grammar and is denoted $L(G)$.*

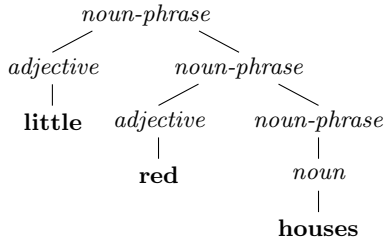


Figure 9.4 The parse tree for the sentence **little red houses**.

Processes that obtain one string of symbols from another by repeated applications of productions are called *derivations*. To indicate that **little red** *noun* can be derived from *noun-phrase*, we write

$$\textit{noun-phrase} \stackrel{*}{\Rightarrow} \textbf{little red } \textit{noun}.$$

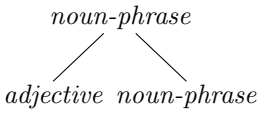
Thus

$$\textit{sentence} \stackrel{*}{\Rightarrow} \textbf{big brown small houses sees green green boys}.$$

We can represent productions by RP-trees where a vertex is the left side of a production and the leaves are the items on the right side; for example,

$$\textit{noun-phrase} \rightarrow \textit{adjective noun-phrase}$$

becomes



The collection of productions thus become local descriptions for trees corresponding to derivations such as that shown in Figure 9.4. We call such a tree a *parse tree*. The string that was derived from the root of the parse tree is the DFV sequence of the tree. A sentence corresponds to a parse tree in which the root is the start symbol and all the leaves are terminal symbols.

How is Figure 9.3 related to parse trees? There certainly seems to be some similarity, but all the symbols are terminals. What has happened is that the parse tree has been squeezed down to eliminate unnecessary information. Before we can think of Figure 9.3 as coming from a parse tree, we need to know what the grammar is. Here's a possibility, with *exp* (short for *expression*) the start symbol, **number** standing for any number and **id** standing for any *identifier*; i.e., the name of a variable.

$$\begin{array}{rcl} \textit{exp} & \rightarrow & \textit{term} \mid \textit{term op term} \\ \textit{term} & \rightarrow & (\ \textit{exp}\ ) \mid \textbf{id} \mid \textbf{number} \\ \textit{op} & \rightarrow & + \mid - \mid * \mid / \end{array}$$

A computer language has a grammar. The main purpose of a compiler is to translate grammatically correct code into machine code. A secondary purpose is to give the programmer useful messages when nongrammatical material is encountered. Whether grammatically correct statements make sense in the context of what the programmer wishes to do is beyond the ken of a compiler; that is, a compiler is concerned with syntax, not with content.

Compilers must use grammars backwards from the way we've been doing it. Suppose you are functioning as a general purpose compiler. You are given the grammar and a string of terminal symbols. Instead of starting with the start symbol, you must start with the string of terminals and works backwards, creating the parse tree *from* the terminals at the leaves *back to* the start symbol at the root. This is called *parsing*. A good compiler must be able to carry out this process or something

like it quite rapidly. For this reason, attention has focused on grammars which are quickly parsed and yet flexible enough to be useful as computer languages.

When concerned with parsing, one should read the productions backwards. For example,

$$\textit{term} \quad \rightarrow \quad (\ \textit{exp}\ ) \quad | \quad \textbf{id} \quad | \quad \textbf{number}$$

would be read as the three statements:

- If one sees the string "(*exp*)", it may be thought of as a *term*.
- If one sees an **id**, it may be thought of as a *term*.
- If one sees a **number**, it may be thought of as a *term*.

Example 9.7 Arithmetic expressions again Our opening example on arithmetic expressions had a serious deficiency: Parentheses were required to group things. We would like a grammar that would obey the usual rules of mathematics: Multiplication and division take precedence over addition and subtraction and, in the event of a draw, operations are performed from left to right.

We can distinguish between factors, terms (products of factors) and expressions (sums of terms) to enforce the required precedence. We can enforce the left to right rule by one of two methods:

(a) We can build it into the syntax.

(b) We can insist that in the event of an ambiguity the leftmost operation should be performed.

The latter idea has important ramifications that make parsing easier. You'll learn about that when you study compilers. We'll use method (a). Here's the productions, with *exp* the start symbol.

$$\begin{array}{rcl} \textit{exp} & \rightarrow & \textit{term} \quad | \quad \textit{exp} + \textit{term} \quad | \quad \textit{exp} - \textit{term} \\ \textit{term} & \rightarrow & \textit{factor} \quad | \quad \textit{term} * \textit{factor} \quad | \quad \textit{term} \,/\, \textit{factor} \\ \textit{factor} & \rightarrow & (\ \textit{exp}\ ) \quad | \quad \textbf{id} \quad | \quad \textbf{number} \end{array}$$

As you can see, even this a language fragment is a bit complicated. □

By altering (4) of Definition 9.2 (p. 254), we can get other types of grammars. A more general replacement rule is

4′. a finite set of *productions* of the form $v_1 \dots v_m \rightarrow x_1 \dots x_n$ where $v_i, x_i \in S \cup T$, $m \geq 1$ and $n \geq 0$.

This gives what are called *phrase structure grammars*. Grammars that are this general are hard to handle.

A much more restrictive replacement rule is

4″. a finite set of *productions* of the form $s \rightarrow \mathbf{t_1} \dots \mathbf{t_n}$ or $s \rightarrow \mathbf{t_1} \dots \mathbf{t_n} r$ and where $r,s \in S$, $n \geq 0$ and $\mathbf{t_i} \in T$.

This gives what are called *regular grammars*, which are particularly easy to study.

Example 9.8 Finite automata and regular grammars In Section 6.6 we studied finite automata and, briefly, Turing machines. If $\mathcal{A}$ is a finite automaton, the *language* of $\mathcal{A}$, $L(\mathcal{A})$, is the set of all input sequences that are recognized by $\mathcal{A}$. With a proper definition of recognition for Turing machines, the languages of phrase structure grammars are precisely the languages recognized by Turing machines. We will prove the analogous result for regular grammars:

Theorem 9.1 Regular Grammars *Let L be a set of strings of symbols. There is a regular grammar G with $L(G) = L$ if and only if there is a finite automaton $\mathcal{A}$ with $L = L(\mathcal{A})$. In other words, the languages of regular grammars are precisely the languages recognized by finite automata.*

Proof: Suppose we are given an automaton $\mathcal{A} = (S, I, f, s_0, A)$. We must exhibit a regular grammar G with $L(G) = L(\mathcal{A})$. Here it is.

1. The set of nonterminal symbols of G is S, the set of states of $\mathcal{A}$.
2. The set of terminal symbols of G is I, set of input symbols of $\mathcal{A}$.
3. The start symbol of G is s_0, the start symbol of $\mathcal{A}$.
4. The productions of G are all things of the form $s \to it$ or $s \to j$ where $f(s, i) = t$ or $f(s, j) \in A$, the accepting states of $\mathcal{A}$.

Clearly G is a regular grammar. It is not hard to see that $L(G) = L(\mathcal{A})$.

Now suppose we are given a regular grammar G. We must exhibit an automaton $\mathcal{A}$ with $L(\mathcal{A}) = L(G)$. Our proof is in three steps. First we show that there is a regular grammar G' with $L(G') = L(G)$ and with no productions of the form $s \to t$. Second we show that there is a regular grammar G'' with $L(G'') = L(G')$ in which the right side of all productions are either empty or of the form $\mathbf{i}s$, that is $\mathbf{i}$ is terminal and s is nonterminal. Finally we show that there is a nondeterministic finite automaton $\mathcal{N}$ that recognizes precisely $L(G'')$. By the "no free will" theorem in Section 6.6, this will complete the proof.

First step: Suppose that G contains productions of the form $s \to t$. We will construct a new regular grammar G' with no productions of this form and $L(G) = L(G')$. (If there are no such productions, simply let $G' = G$.) The terminal, nonterminal and start symbols of G' are the same as those of G. Let R be the right side of a production in G that is not simply a nonterminal symbol. We will let $s \to R$ be a production in G' if and only if

- $s \to R$ is a production in G, or
- There exist $x_1, \ldots, x_n$ such that $s \to x_1$, $x_1 \to x_2$, $\cdots$, $x_{n-1} \to x_n$ and $x_n \to R$ are all productions in G.

You should be able to convince yourself that $L(G') = L(G)$.

Second step: We now construct a regular grammar G'' in which the right side of every production is either empty or has exactly one terminal symbol. The terminal symbols of G'' are the same as those of G'. The nonterminal symbols of G'' are those of G' plus some additional ones which we'll define shortly.

Let $s \to \mathbf{i}_1 \ldots \mathbf{i}_n t$ be a production in G'. If $n = 1$, this is also a production in G''. By the construction of G', we cannot have $n = 0$, so we can assume $n > 1$. Let σ stand for the right side of the production. Introduce $n - 1$ new states (s, σ, k) for $2 \le k \le n$. Let G'' contain the productions

- $s \to \mathbf{i}_1(s, \sigma, 2)$;
- $(s, \sigma, k) \to \mathbf{i}_k(s, \sigma, k + 1)$, for $1 < k < n$ (There are none of these if $n = 2$.);
- $(s, \sigma, n) \to \mathbf{i}_n t$.

Let $s \to \mathbf{i}_1 \ldots \mathbf{i}_n$ be a production in G'. (An empty right hand side corresponds to $n = 0$.) If $n = 0$ or $n = 1$, let this be a production in G''; otherwise, use the idea of the previous paragraph, omitting t. You should convince yourself that $L(G'') = L(G')$.

Third step: We now construct a nondeterministic finite automaton $\mathcal{N}$ that recognizes precisely $L(G)$. The states of $\mathcal{N}$ are the internal symbols of G'' together with a new state a, the start state is the start symbol, and the input symbols are the terminal symbols. Let a be an accepting state of $\mathcal{N}$. Let $s \to R$ be a production of G''. There are three possible forms for R:

- If R is empty, then s is an accepting state of $\mathcal{N}$.
- If $R = \mathbf{i}$, then $(s, \mathbf{i}, a)$ is an edge of $\mathcal{N}$.
- if $R = \mathbf{i}t$, then $(s, \mathbf{i}, t)$ is an edge of $\mathcal{N}$.

You should convince yourself that $\mathcal{N}$ accepts precisely $L(G'')$. □

Exercises

9.2.1. Draw the trees like Figure 9.3(c) to interpret the following expressions.

(a) $(((1+2)+3)+4)+5$

(b) $((1+2)+(3+4))+5$

(c) $1+(2+(3+(4+5)))$

(d) $(X+5*Y)/(X-Y)$

(e) $(X*Y-3)+X*(Y+1)$

9.2.2. How might the ideas in Example 9.5 be modified to allow for unary minus as in $(-A)*B$?

9.2.3. Write pseudocode for the two methods suggested in Example 9.5 for calculating the value of an arithmetic expression.

9.2.4. Using the grammar of Example 9.7, construct parse trees for the following sentences.

(a) $(X+5*Y)/(X-Y)$

(b) $(X*Y-3)+X*(Y+1)$

9.2.5. Add the following features to the expressions of Example 9.7.

(a) *Unary minus.*

(b) *Exponentiation* using the operator $**$. Ambiguities are resolved in the reverse manner from the other arithmetic operations: **A** $**$ **B** $**$ **C** is the same as **A** $**$ **(B** $**$ **C)**.

(c) *Replacement* using the operator **:=**. Only one use of the operator is allowed.

(d) *Multiple replacement* as in

$$\mathbf{A := 4 + B := C := 5 + 2 * 3},$$

which means that C is set equal to $5+2*3$, B is set to equal to C and A is set equal to $4+B$.

9.2.6. Let G be the grammar with the start state s and the productions

(i) $s \to \mathbf{x}t$ and $s \to \mathbf{y}t$;

(ii) $t \to R$ where R is either empty or one of $+\mathbf{x}t$, $+\mathbf{y}t$ or $-\mathbf{x}t$.

(a) Describe $L(G)$.

(b) Follow the steps in the construction of the corresponding nondeterministic finite automaton; that is, describe G', G'' and $\mathcal{N}$ that were constructed in the proof.

(c) Continuing the previous part, construct a deterministic machine as done in Section 6.6 corresponding to $\mathcal{N}$.

(d) Can you construct a simpler deterministic machine to recognize $L(G)$?

*9.3 Unlabeled Full Binary RP-Trees

We'll begin with a review of material discussed in Examples 7.9 (p. 206) and 7.10. Roughly speaking, an unlabeled RP-tree is an RP-tree with the vertex labels erased. Thus, the order of the sons of a vertex is still important. A tree is "binary" (resp. "full binary") if each nonleaf has at most (resp. exactly) two sons. Figure 9.5 shows some unlabeled full binary RP-trees. Here is a more precise pictorial definition. Compare it to Definition 9.1 for (labeled) RP-trees.

Definition 9.4 Unlabeled binary rooted plane trees *The following are* **unlabeled binary RP-trees**. *Roots are indicated by* • *and other vertices by* ∘.

(*i*) *The single vertex* • *is such a tree.*

(*ii*) *If* T_1 *is one such tree, so is the tree formed by (a) drawing* T_1 *root upward, (b) adding a* • *above* T_1 *and connecting* • *to the root of* T_1*, and (c) changing the root of* T_1 *to* ∘.

(*iii*) *If* T_1 *and* T_2 *are two such trees, so is the tree formed by (a) drawing* T_1 *to the left of* T_2*, both root upward, (b) adding a* • *above them and connecting it to their roots, and (c) changing the roots of* T_1 *and* T_2 *to* ∘*'s.*

If we omit (ii), the result is **unlabeled full binary RP-trees**.

These trees are often referred to as "unlabeled ordered (full) binary trees." Why? To define a binary tree, one needs to have a root. Drawing a tree in the plane is equivalent to ordering the children of each vertex. Sometimes the adjective "full" is omitted. In this section, we'll study unlabeled ordered full binary trees.

We can build all unlabeled full binary RP-trees recursively by applying the definition over and over. To begin with there are no trees, so all we get is a single vertex by (1) of the definition. This tree can then be used to build the 3 vertex full binary RP-tree shown in the next step of Figure 9.5. Using the two trees now available, we can build the three new trees shown in the right hand step of Figure 9.5. In general, if we have a total of t_n trees at step n, then $t_1 = 1$ (the single vertex) and $t_{n+1} = 1 + (t_n)^2 + 1$ (either use the single vertex tree or join two trees T_1 and T_2 to a new root).

Example 9.9 Counting and listing unlabeled full binary RP-trees How many unlabeled full binary RP-trees are there with n leaves? How can we list them?

As we shall see, answers to these questions come almost immediately from the recursive definition. It is important to note that

Definition 9.4 provides *exactly one way* to produce every unlabeled full binary RP-tree.

If there were more than one way to produce some of the trees from the definition, we would not be able to obtain answers to our questions so easily, if at all.

We begin with counting. Let b_n be the desired number. Clearly $b_0 = 0$, since a tree has at least one leaf. Let's look at how our definition leads to trees with n leaves.

According to the definition, an unlabeled full binary RP-tree will be either a single vertex, which contributes to b_1, or it will have exactly two principal subtrees, both of which are unlabeled full binary RP-trees. If the first of these has k leaves, then the second must have $n-k$. By the Rules of Sum and Product,

$$b_n = \sum_{k=1}^{n-1} b_k b_{n-k} \qquad \text{if } n > 1. \tag{9.3}$$

Using this we can calculate the first few values fairly easily:

$$b_1 = 1 \qquad b_2 = 1 \qquad b_3 = 2 \qquad b_4 = 5 \qquad b_5 = 14 \qquad b_6 = 42 \qquad b_7 = 132.$$

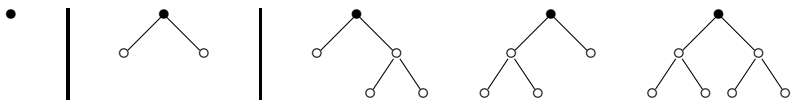


Figure 9.5 The first three stages in building unlabeled full binary RP-trees recursively. A ∘ is a vertex of a previously constructed tree and a • is the root of a new tree.

Notice how the recursion came almost immediately from the definition.

So far, this has all been essentially a review of material in Examples 7.9 and 7.10. Now we'll look at something new: listing the trees based on the recursive description. Here's some pseudocode to list all binary RP-trees with n leaves.

```
/* Make a list of n-leaf unlabeled full binary RP-trees */
Procedure BRPT(n)
     If (n = 1), then Return the single vertex tree
     Set List empty
     For k = 1, 2, ···, n − 1:
          /* Get a list of first principal subtrees. */
          S_L = BRPT(k)
          /* Get a list of second principal subtrees. */
          S_R = BRPT(n − k)
          For each T_1 ∈ S_L:
              For each T_2 ∈ S_R:
                   Add JOIN(T_1, T_2) to List
              End for
          End for
     End for
     Return List
End
```

The procedure JOIN(T_1, T_2) creates a full binary RP-tree with principal subtrees T_1 and T_2. The outer `for` loop is running through the terms in the summation in (9.3). The inner `for` loop is constructing each of the trees that contribute to the product $b_k b_{n-k}$. This parallel between the code and (9.3) is perfectly natural: They both came from the same recursive description of how to construct unlabeled full binary RP-trees. □

What happened in this example is typical. Given a recursive description of how to *uniquely* construct all objects in some set, we can provide both a recursion for the number of them and pseudocode to list all of them. It is not so obvious that such a description usually leads to ranking and unranking algorithms as well. Rather than attempt a theoretical explanation of how to do this, we'll look at examples. Since it's probably not fresh in your mind, it would be a good idea to review the concepts of ranking and unranking from Section 3.2 (p. 75) at this time.

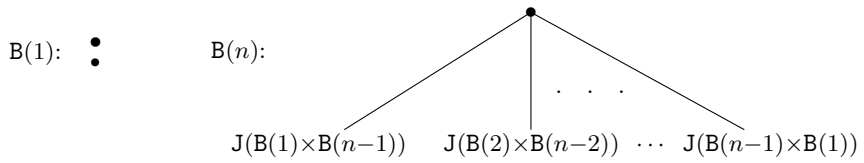


Figure 9.6 The local description for n-leaf unlabeled full binary RP-trees. We assume that $n > 1$ in the right hand figure. `B` stands for `BRPT` and `J(C,D)` $= \{$`JOIN`$(c, d) \mid c \in$ `C`, $d \in$ `D`$\}$, with the set made into an ordered list using lexicographic order based on the orderings in `C` and `D` obtained by lex ordering all the pairs $(\mathrm{RANK}(c), \mathrm{RANK}(d))$, with $c \in$ `C` and $d \in$ `D`.

Example 9.10 A ranking for permutations

We'll start with permutations since they're fairly simple.

Suppose that S is an n-set with elements $s_1 < \cdots < s_n$. The local description of how to generate $L(S)$, the permutations of S listed in lex order, is given in Figure 7.2 (p. 213). This can be converted to a verbal recursive description: Go through the elements $s_i \in S$ in order. For each s_i, list all the permutations of $S - \{s_i\}$ in lex order, each preceded by "s_i,". (The comma after s_i is not a misprint.)

How does this lead to RANK and UNRANK functions? Let $\sigma\colon \underline{n} \to \underline{n}$ be a permutation and let $\mathrm{RANK}(s_{\sigma(1)}, \ldots, s_{\sigma(n)})$ denote the rank of $s_{\sigma(1)}, \ldots, s_{\sigma(n)}$. Since the description is recursive, the rank formula will be as well. We need to start with $n = 1$. Since there is only one permutation of a one-element set, $\mathrm{RANK}(\sigma) = 0$.

Now suppose $n > 1$. There are $\sigma(1) - 1$ principal subtrees of $L(S)$ to the left of the subtree

$$s_{\sigma(1)}, L(S - \{s_{\sigma(1)}\}),$$

each of which has $(n-1)!$ leaves. Thus we have

$$\mathrm{RANK}(s_{\sigma(1)}, \ldots, s_{\sigma(n)}) \;=\; (\sigma(1) - 1)(n-1)! + \mathrm{RANK}(s_{\sigma(2)}, \ldots, s_{\sigma(n)}).$$

You should also be able to see this by looking at Figure xrefLexOrderLocal.

As usual the rank formula can be "reversed" to do the unranking: Let $\mathrm{UNRANK}(r, S)$ denote the permutation of the set S that has rank r. Let $q = r/(n-1)!$ with the remainder discarded. Then

$$\mathrm{UNRANK}(r, S) \;=\; s_{q+1},\ \mathrm{UNRANK}\big(r - (n-1)!q,\ S - \{s_{q+1}\}\big). \quad \square$$

Example 9.11 A ranking for unlabeled full binary RP-trees

How can we rank and unrank `BRPT`(n), the n-leaf unlabeled full binary RP-trees?

Either Definition 9.1 or the listing algorithm `BRPT` that we obtained from it in Example 9.9 can be used as a starting point. Each gives a local description of a decision tree for generating n-leaf unlabeled full binary RP-trees. The only thing that is missing is an ordering of the sons in the decision tree, which can be done in any convenient manner. The listing algorithm provides a specific order for listing the trees, so we'll use it. It's something like lex order:

- first by size of the left tree (the outer loop on k),
- then by the rank of the left tree (the middle loop on $T_1 \in S_L$), 9.4
- finally by the rank of the right tree (the inner loop on $T_2 \in S_R$).

The left part of the figure is not a misprint: the top • is the decision tree and the bottom • is it's label: the 1-leaf tree. Carrying this a bit further, Figure 9.7 expands to local description for 2 and 3 leaves.

Expanding this local description for any value of n would give the complete decision tree in a nonrecursive manner. However, we have learned that expanding recursive descriptions is usually unnecessary and often confusing. In fact, we can obtain ranking and unranking algorithms directly from the local description, as we did for permutations in the preceding example.

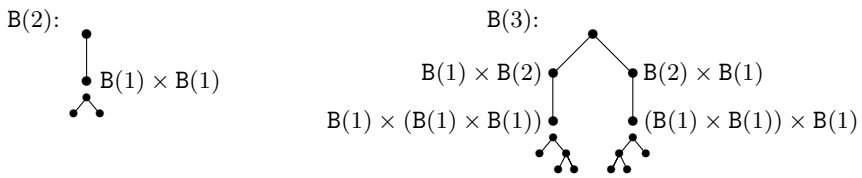


Figure 9.7 An expansion of Figure 9.6 for $n = 2$ and $n = 3$. The upper trees are the expansion of Figure 9.6. The lower trees are the full binary RP-trees that occur at the leaves.

Let's get a formula for the rank. Since our algorithm for listing is recursive, our rank formula will also be recursive. We must start our recursive formula with the smallest case, $|T| = 1$. In this case there is only one tree, namely a single vertex. Thus $T = \bullet$ and $\mathrm{RANK}(T) = 0$.

Now suppose $|T| > 1$. and let T_1 and T_2 be its first and second principal subtrees. (Or left and right, if you prefer.) We need to know which trees come before T in the ranking. Suppose Q has principal subtrees Q_1 and Q_2 and $|Q| = |T|$. The information in (9.4) says that the tree T is preceded by precisely those trees Q for which $|Q| = |T|$, and either

- $|Q_1| < |T_1|$ OR
- $|Q_1| = |T_1|$ AND $\mathrm{RANK}(Q_1) < \mathrm{RANK}(T_1)$ OR
- $Q_1 = T_1$ AND $\mathrm{RANK}(Q_2) < \mathrm{RANK}(T_2)$.

The number of trees in each of these categories is

- $\displaystyle\sum_{k<|T_1|} b_k b_{n-k}$, where terms were collected by $k = |Q_1|$,
- $\mathrm{RANK}(T_1) \times b_{|T_2|}$, and
- $1 \times \mathrm{RANK}(T_2)$.

Hence

Theorem 9.2 Rank of Unlabeled Full Binary RP-Trees *The rank of an unlabeled full binary RP-tree with n leaves is 0 if $n = 1$ and otherwise is*

$$\mathrm{RANK}(T) \;=\; \sum_{k<|T_1|} b_k b_{n-k} + \mathrm{RANK}(T_1) b_{|T_2|} + \mathrm{RANK}(T_2), \tag{9.5}$$

where T_1 and T_2 are the first and second principal subtrees of T and b_k is number of k-leaf unlabeled full binary RP-trees.

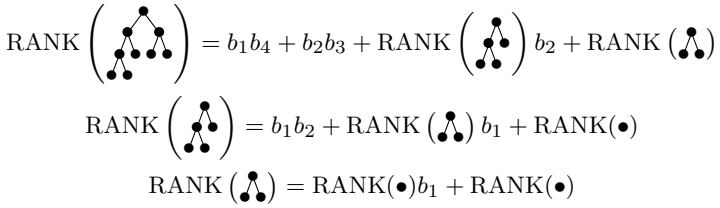


Figure 9.8 A recursive rank calculation for an unlabeled full binary RP-tree with 5 leaves.

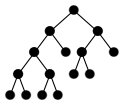


Figure 9.9 The 8-leaf unlabeled full binary RP-tree of rank 250.

Figure 9.8 shows how (9.5) is used to compute rank. Each of the equations given there is a special case of (9.5). The first equation gives the rank of the tree we are interested in. The other two equations give ranks that are needed because of the recursive nature of (9.5). One can now work from the bottom up using $\mathrm{RANK}(\bullet) = 0$ to get ranks of 0, 1 and 8, respectively.

As always, unranking uses a greedy algorithm. Let $\mathrm{UNRANK}(R, n)$ denote the n-leaf full binary RP-tree with rank R. Let's compute $\mathrm{UNRANK}(250, 8)$, the 8-leaf full binary RP-tree with rank 250. Being greedy, we want T_1, the left principal tree to have as many leaves as possible. We have

$$b_1b_7 + b_2b_6 + b_3b_5 + b_4b_4 \;=\; 227$$

and

$$b_1b_7 + b_2b_6 + b_3b_5 + b_4b_4 + b_5b_3 \;=\; 227 + 28 > 250.$$

Thus the first principal tree, T_1, has five leaves and the second, T_2, has three. Now we want $\mathrm{RANK}(T_1)$ to be as large as possible. Since $\mathrm{RANK}(T_1)b_3 + \mathrm{RANK}(T_2) = 250 - 227 = 23$ and $23/b_3 = 23/2 = 11.5$, T_1 has rank 11 and T_2 has rank $23 - 11b_3 = 1$. Thus $T_1 = \mathrm{UNRANK}(11, 5)$ and $T_2 = \mathrm{UNRANK}(1, 3)$. We'll compute T_2 first. We have $b_1b_2 = 1$ and $b_2b_1 = 1$, so the first principal subtree of T_2 has two leaves and the second has one. Since there is only one 2-leaf tree and only one 1-leaf, we are done with T_2. Since $b_1b_4 + b_2b_3 + b_3b_2 = 9$, the first principal subtree of T_1 has four leaves and rank 2 while the second has one leaf. Since $b_1b_3 = 2$, both principal subtrees of this 4-leaf tree have two leaves. Putting this all together, we get the tree shown in Figure 9.9. □

Example 9.12 Computing the rank without recursion In Example 9.11 (p. 261) we proved the recursive formula (9.5) for the rank of an unlabeled full binary RP-tree. This formula can be implemented as it stands by a recursive computer program; however, recursive procedures can be inconvenient for hand calculations. We can implement the formula by a depth first postorder vertex traversal of the tree that we want to rank.

The last time we visit a vertex, we simply record the rank and number of leaves in the subtree which has that vertex as its root. If we are at a leaf, the rank is 0 and the number of leaves is 1. Suppose we have reached some tree T that is not a leaf. In the notation of (9.5) we have available $\mathrm{RANK}(T_1)$, $|T_1|$, $\mathrm{RANK}(T_2)$ and $|T_2|$. From this we can compute $\mathrm{RANK}(T)$ using (9.5) because $n = |T_1| + |T_2|$. The values of the b_i's can be computed ahead of time and written in a table. Figure 9.10 applies this idea to the tree in Figure 9.9. Rather than work in depth first postorder, we can simply do the vertices depth by depth, starting at the lowest depth. □

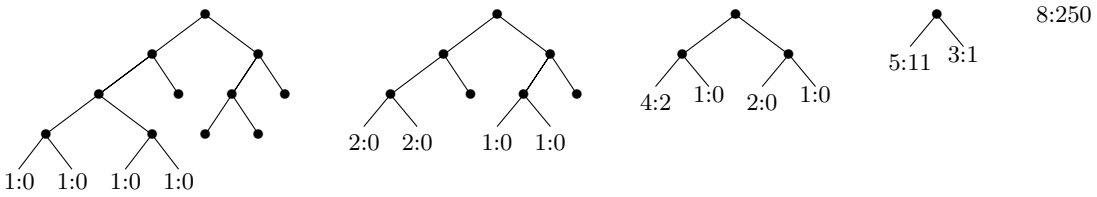


Figure 9.10 Computing the rank of the tree in Figure 9.9. As we move up in the tree, we replace a vertex v with $L{:}\,R$ where R is the rank of the subtree rooted at v and L is how many leaves it has. When information is no longer needed, we discard it to keep the figures from getting cluttered.

Example 9.13 Calculating statistics for RP-trees When RP-trees are used as data structures, items of data may be stored at the leaves and an "action" such as finding an item in a list may involve finding the appropriate leaf by traversing the path from the root to the leaf. How fast can data in such a tree be accessed?

The tree is being used as a decision tree and the number of decisions needed to reach the leaf equals the length of the path. Thus, the time needed to find an item this way is usually nearly proportional to the length of the path traversed. Given an RP-tree T, the length of the longest path from the root to a leaf, $m(T)$, say, and the average over all leaves of the lengths of the paths, $\mu(T)$, say, are therefore important measures of how good the tree is for storing data. Worst case (i.e., longest) time is proportional to $m(T)$ and average time to $\mu(T)$.

Given a particular tree T, we could calculate $m(T)$ and $\mu(T)$. Suppose we are told that the algorithm for creating the data structure constructs a random tree from some class; e.g., the set of n-leaf unlabeled full binary RP-trees. How can we get information about $\mu(T)$ in this case? Here are some possible approaches assuming we are dealing with unlabeled full binary RP-trees.

- Average over all: We might try to compute the average value of $\mu(T)$ over all such trees, provided we have a way to list all of them. Call the average value $\mu(n)$. We could then compute $\mu(T)$ for each tree T and average the results to obtain $\mu(n)$. If we are lucky enough to have an unranking algorithm, we can list all the trees using `UNRANK`(i, n) for $i = 0, 1, \ldots, b_n - 1$. Unfortunately, b_n is much too large for realistic values of n, so the program would take too long to run.
- Generate some at random and average: Another method is to generate n-leaf unlabeled full binary RP-trees at random by the method mentioned at the start of Section 3.2: Choose a random integer in $[0, b_n)$, unrank it and study the result. Repeat this procedure many times to get a good estimate. Choosing many elements from a set at random and studying them is known as the *Monte Carlo method* or *Monte Carlo simulation*. Studying it would take us too far afield.
- Use generating functions: We might try to find a theoretical tool. Indeed, we'll be able to compute $\mu(n)$ using generating functions (Exercise 11.2.16 (p. 329)). Theoretical methods can be wonderful when they work, but they have a nasty habit of not working when we change the problem. For example, our method will not give us the average of $m(T)$, the length of the longest path to the root. It can be estimated theoretically, but it is far more difficult than determining the average of $\mu(T)$.

Suppose we are looking at structures where we don't have an unranking algorithm and we can't afford to list all of them. It appears that we must use generating functions. This is not necessarily the case. Suppose that we have an unranking algorithm for a set that contains the one we are interested in and is not too much larger. We can use that algorithm, rejecting completely those structures that lie outside the set of interest. Here is a general pseudocode procedure for this method.

```
Procedure ESTIMATE(setsize,ncases,parameters)
     Initialize
     /* Loop to generate ncases examples. */
     For i = 1, 2, ..., ncases:
          /* Need a case in set of interest. */
          Set needcase to true
          While needcase is true:
               Choose a random integer j ∈ [0, setsize)
               T = UNRANK(j, parameters)
               If T is okay, then set needcase to false
          End while
          Store information about T as desired
     End for
     Finish it up: calculate and output concluding information
End
```

We will not study such algorithms in this text. □

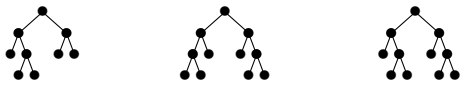
Exercises

9.3.1. Using Definition 9.1, recursively construct *all* unlabeled RP-trees with at most 4 vertices. Note that you are not supposed to simply list them. You should iterate the definition and retain all trees of at most 4 vertices that arise. When the list does not change during an iteration, it is complete.
Note: You are asked for all trees, not just the binary ones.

9.3.2. Using Definition 9.4, recursively construct all unlabeled full *binary* RP-trees with at most 4 leaves.

9.3.3. Construct a table of b_n for $n \leq 10$.

9.3.4. Compute the ranks of the unlabeled full binary RP-trees shown here.



9.3.5. Construct the unlabeled full binary RP-trees with eight leaves whose ranks are 100, 200, 300 and 400.

9.3.6. Prove that a full binary RP-tree with n leaves has $n-1$ other vertices.

9.3.7. We are interested in the unlabeled full binary RP-tree with n leaves and rank $b_n/2$; i.e., the tree just past the middle of the list. Call the tree $\mathcal{M}_n$.

(a) Construct $\mathcal{M}_3$, $\mathcal{M}_5$ and $\mathcal{M}_7$.

(b) Conjecture and prove the nature of $\mathcal{M}_n$ when n is odd.

(c) Conjecture and prove the nature of $\mathcal{M}_n$ when n is even.

9.3.8. Provide a recursive method for calculating the rank of a decreasing function in lex order.

9.3.9. Use equivalence relations to provide a formal definition for unlabeled RP-trees in terms of labeled RP-trees.

9.3.10. Provide a recursive method for calculating the lex order rank of a permutation.

9.3.11. Let $**$ stand for the binary operation of exponentiation. How parentheses are placed in the expression $x{**}y{**}z$ effects the answer. Thus $3{**}(2{**}3) = 3^{2^3} = 3^8 = 1458$ while $(3{**}2){**}3 = (3^2)^3 = 3^6 = 729$. We would like to generate all ways of parenthesizing $x_1 {**} \cdots {**} x_n$. This can be done by *first* selecting the *last* $**$ operation to be performed as in

$$(x_1 ** \cdots ** x_k) ** (x_{k+1} ** \cdots ** x_n),$$

and then proceeding recursively on $x_1 {**} \cdots {**} x_k$ and $x_{k+1} {**} \ldots {**} x_n$. (If $k = 1$ we have simply (x_1) on the left.) The recursion stops when every innermost pair of parentheses contains just one number as in (x_i). Call this final result a "parenthesizing."

(a) Show that if you remove the x_i's from a parenthesizing, it is possible to tell where they belong. Thus all we need are the parentheses.

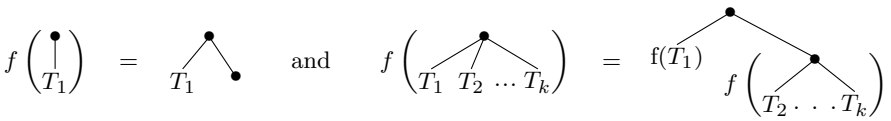
(b) Show that the set of possible parentheses patterns leads to a tree that looks the same as that in Figure 9.6 with $\bullet$ replaced by ( ) and $\texttt{JOIN}(A, B)$ interpreted as (AB).

(c) It follows from (b) that there is a simple correspondence between the parenthesized expressions and unlabeled full binary RP-trees. Describe it.

9.3.12. If T_i are unlabeled RP-trees, let $[T_1, \ldots, T_k]$ denote the unlabeled rooted RP-tree in which the ith edge from the root leads to the root of T_i. In particular, [ ] is the tree $\bullet$. We define a map f from the unlabeled RP-trees to the unlabeled full binary RP-trees recursively as follows. Let $f\big([\,]\big) = [\,] = \bullet$ and

$$f\big([T_1, \ldots, T_k]\big) \;=\; \Big[f(T_1),\; f\big([T_2, \ldots, T_k]\big)\Big] \tag{9.6}$$

when $k > 0$. Pictorially,



(a) Show that f is a bijection between n-vertex unlabeled RP-trees and n-leaf unlabeled full binary RP-trees.

(b) Use the above correspondence to find a procedure for ranking and unranking the set of all n-vertex unlabeled RP-trees. Provide a local description like Figure 9.6.

*9.3.13. In this exercise you will obtain a formula for b_n by proving a simple recursion. You might ask "How would I be expected to discover such a result?" Our answer at this time would be "Luck and/or experience." When we study generating functions, you'll have a more systematic method.

Let $\mathcal{L}_n$ be the set of n-leaf unlabeled full binary RP-trees with one leaf marked and let $\mathcal{V}_n$ with the set with one vertex marked.

(a) Prove that $\mathcal{L}_n$ has nb_n elements and that $\mathcal{V}_n$ has $(2n-1)b_n$ elements.

(b) Consider the following operation on each element of $\mathcal{L}_{n+1}$. If x is the marked leaf, let f be its father and b its brother. Remove x and shrink the edge between f and b so that f and b merge into a single vertex. Prove that each element of $\mathcal{V}_n$ arises exactly twice if we do this with each element of $\mathcal{L}_{n+1}$.

(c) From the previous results, conclude that $(n+1)b_{n+1} = 2(2n-1)b_n$.

(d) Use the recursion just derived to obtain a fairly simple formula for b_n.

Notes and References

Books on combinatorial algorithms and data structures usually discuss trees. You may wish to look at the references at the end of Chapter 6. Grammars are discussed extensively in books on compiler design such as [1].

1. Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley (1977).

PART IV

Generating Functions

Suppose we are given a sequence $a_0, a_1, \ldots$. The "ordinary generating function" associated with the sequence $a_0, a_1, \ldots$ is the function $A(x)$ whose value at x is the power series $\sum_{i \geq 0} a_i x^i$. In other words,"ordinary generating function of" is a map (function) from sequences to power series that "packages" the entire series of numbers $a_0, a_1, \ldots$ into a single function $A(x)$. Generating functions are not limited to sequences with single indices; for example, we will see that

$$\sum_{n \geq 0} \sum_{k \geq 0} \binom{n}{k} x^k y^n \;=\; (1 - y(1+x))^{-1}.$$

For simplicity, this introductory discussion is phrased in terms of singly indexed sequences.

It is often easier to manipulate generating functions than it is to manipulate sequences $a_0, a_1, \ldots$. We can obtain information about a sequence through the manipulation of its generating function. We have seen a little of this in Example 1.14 (p. 19) and in Section 1.5 (p. 36). In the next two chapters, we will study generating functions in more detail.

What sort of information is needed about a sequence $a_0, a_1, \ldots$, to obtain $A(x)$, the generating function for the sequence? Of course, an explicit formula for a_n as a function of n would provide this but there are often better methods:

- Recursions: A recursion for the a_n's may give an equation that can be solved for $A(x)$. We'll study this in Sections 10.2 and 11.1.
- Constructions: A method for constructing the objects counted by the a_n's may lead to an equation for $A(x)$. Sometimes this can be done using an extension of the Rules of Sum and Product. See Sections 10.4 and 11.2.

Given a generating function $A(x)$, what sort of information might we obtain about the sequence $a_0, a_1, \ldots$ associated with it?

- An explicit formula: *Taylor's Theorem* from calculus is an important tool. It tells us that $A(x) = \sum \big(A^{(n)}(0)/n!\big)x^n$, and so gives us a formula if $A(x)$ is simple enough. In particular, we easily obtain a rather simple formula for b_n, the number of unlabeled binary trees with n leaves. In Section 9.3 we obtained only a recursion.
- A recursion: Simply equate coefficients in an equation that involves $A(x)$. This is done in various places in the following chapters, but especially in Section 10.3.
- Statistical information: For example, the expected number of cycles in a random permutation of $\underline{n}$ is approximately $\ln n$. The degree of the root of a random labeled RP-tree is approximately one more than a "Poisson random variable," a subject beyond the scope of this text.
- Asymptotic information: In other words, how does a_n behave when n is large? Methods for doing this are discussed in Section 11.4.
- Prove identities: We saw some of this in Section 1.5 (p. 36).

We hope that the preceding list has convinced you that generating functions are an important tool that yield results that are sometimes difficult (or perhaps even impossible) to obtain by other

methods. If you find generating functions difficult to understand, keep this motivation in mind as you study them.

The next chapter introduces the basic concepts associated with generating functions:

- Basic concepts: We define a generating function and look at some basic manipulations.
- Recursions: If you have encountered recursions in other courses or do not wish to study them, you can skim Section 10.2. However, if you had difficulty in Section 10.1, you should use this section to obtain additional practice with generating functions.
- Manipulations: In Section 10.3 we present some techniques for manipulating generating functions.
- Rule of Sum and Product: We consider Section 10.4 to be the heart of this chapter. In it, we extend the definition of generating function a bit and obtain Rules of Sum and Product which do for generating functions what the rules with the same name did for basic counting in Chapter 1.

In Chapter 11 we take up four separate topics:

- Systems of recursions: This is a continuation of the discussion in Section 10.2.
- Exponential generating functions: These play the same role for objects with labels that ordinary generating functions play for unlabeled objects in Section 10.4.
- Counting objects with symmetries: We apply generating functions to the problems discussed in Section 4.3 (p. 111).
- Asymptotics: We discuss methods for obtaining asymptotics from generating functions and, to a lesser extent, from recursions.

The sections in Chapter 11 can be read independently of one another; however, some of the asymptotic examples make use of results (but not methods) from Section 11.2.

CHAPTER 10

Ordinary Generating Functions

Introduction

We'll begin this chapter by introducing the notion of ordinary generating functions and discussing the basic techniques for manipulating them. These techniques are merely restatements and simple applications of things you learned in algebra and calculus. You *must* master these basic ideas before reading further.

In Section 2, we apply generating functions to the solution of simple recursions. This requires no new concepts, but provides practice manipulating generating functions. In Section 3, we return to the manipulation of generating functions, introducing slightly more advanced methods than those in Section 1. If you found the material in Section 1 easy, you can skim Sections 2 and 3. If you had some difficulty with Section 1, those sections will give you additional practice developing your ability to manipulate generating functions.

Section 4 is the heart of this chapter. In it we study the Rules of Sum and Product for ordinary generating functions. Suppose that we are given a combinatorial description of the construction of some structures we wish to count. These two rules often allow us to write down an equation for the generating function directly from this combinatorial description. Without such tools, we may get bogged down in lengthy algebraic manipulations.

10.1 What are Generating Functions?

In this section, we introduce the idea of ordinary generating functions and look at some ways to manipulate them. This material is essential for understanding later material on generating functions. Be sure to work the exercises in this section before reading later sections!

Definition 10.1 Ordinary generating function (OGF) *Suppose we are given a sequence* $a_0, a_1, \ldots$. *The* **ordinary generating function** *(also called* **OGF***) associated with this sequence is the function whose value at* x *is* $\sum_{i=0}^{\infty} a_i x^i$. *The sequence* $a_0, a_1, \ldots$ *is called the coefficients of the generating function.*

People often drop "ordinary" and call this the *generating function* for the sequence. This is also called a "power series" because it is the sum of a series whose terms involve powers of x. The summation is often written $\sum_{i\geq 0} a_i x^i$ or $\sum a_i x^i$.

If your sequence is finite, you can still construct a generating function by taking all the terms after the last to be zero. If you have a sequence that starts at a_k with $k > 0$, you can define $a_0, \ldots, a_{k-1}$ to be any convenient values. "Convenient values" are ones that make equations nicer in some sense. For example, if $H_{n+1} = 2H_n + 1$ for $n > 0$ and $H_1 = 1$. It is convenient to let $H_0 = 0$ so that the recursion is valid for $n \geq 0$. (H_n is the number of moves required for the Tower of Hanoi puzzle. See Exercise 7.3.9 (p. 218).) On the other hand, if $b_1 = 1$ and $b_n = \sum_{k=1}^{n-1} b_k b_{n-k}$ for $n > 1$, it's convenient to define $b_0 = 0$ so that we have $b_n = \sum_{k=0}^{n} b_k b_{n-k}$ for $k \neq 1$. (The latter sum is a "convolution", which we will define in a little while.)

To help us keep track of which generating function is associated with which sequence, we try to use lower case letters for sequences and the corresponding upper case letters for the generating functions. Thus we use the function A as generating function for a sequence of a_n's and B as the generating function for b_n's. Sometimes conventional notation for certain sequences make this upper and lower case pairing impossible. In those cases, we improvise.

You may have noticed that our definition is incomplete because we spoke of a function but did not specify its domain or range. The domain will depend on where the power series converges; however, for combinatorial applications, there is usually no need to be concerned with the convergence of the power series. As a result of this, we will often ignore the issue of convergence. In fact, we can treat the power series like a polynomial with an infinite number of terms. The domain in which the power series converges does matter when we study asymptotics, but that is still several sections in the future.

If we have a doubly indexed sequence $b_{i,j}$, we can extend the definition of a generating function:

$$B(x,y) \;=\; \sum_{j\geq 0}\sum_{i\geq 0} b_{i,j} x^i\, y^j \;=\; \sum_{i,j=0}^{\infty} b_{i,j} x^i\, y^j.$$

Clearly, we can extend this idea to any number of indices—we're not limited to just one or two.

Definition 10.2 $[x^n]$ *Given a generating function $A(x)$ we use $[x^n]\,A(x)$ to denote a_n, the coefficient of x^n. For a generating function in more variables, the coefficient may be another generating function. For example $[x^n y^k]\,B(x,y) = b_{n,k}$ and $[x^n]\,B(x,y) = \sum_{i\geq 0} b_{n,i} y^i$.*

Implicit in the preceding definition is the fact that the generating function *uniquely determines* its coefficients. In other words, given a generating function there is just one sequence that gives rise to it. Without this uniqueness, generating functions would be of little use since we wouldn't be able to recover the coefficients from the function alone.

This leads to another question. Given a generating function, say $A(x)$, how can we find its coefficients $a_0, a_1, \ldots$? One possibility is that we might know the sequence already and simply recognize its generating function. Another is Taylor's Theorem. We'll phrase it slightly differently here to avoid questions of convergence. In our form, it is practically a tautology.

Theorem 10.1 Taylor's Theorem *If $A(x)$ is the generating function for a sequence $a_0, a_1, \ldots$, then $a_n = A^{(n)}(0)/n!$, where $A^{(n)}$ is the nth derivative of A and $0! = 1$. (The theorem extends to more than one variable, but we will not state it.)*

We stated this to avoid questions of convergence—but don't we have to worry about convergence of infinite series? Yes and no:
When manipulating generating functions we normally do not need to worry about convergence unless we are doing asymptotics (see Section 11.4) or substituting numbers for the variables (see the next example).

Example 10.1 Binomial coefficients Let's use the binomial coefficients to get some practice. Set $a_{k,n} = \binom{n}{k}$. Remember that $a_{k,n} = 0$ for $k > n$. From the Binomial Theorem, $(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$. Thus $\sum a_{k,n} x^k = (1+x)^n$ and so

$$A(x,y) \;=\; \sum_{n\ge 0}\sum_{k\ge 0} a_{k,n} x^k y^n \;=\; \sum_{n\ge 0} (1+x)^n\, y^n \;=\; \sum_{n=0}^{\infty} ((1+x)y)^n.$$

From the formula $\sum_{k\ge 0} az^k = a/(1-z)$ for summing a *geometric series*, we have

$$A(x,y) \;=\; \frac{1}{1-(1+x)y} \;=\; \frac{1}{1-y-xy}. \tag{10.1}$$

Let's see what we can get from this.

- From our definitions, $[x^k y^n]\, A(x,y) = \binom{n}{k}$ and $[y^n]\, A(x,y) = (1+x)^n$, which is equivalent to

$$\sum_{k=0}^{n} \binom{n}{k} x^k \;=\; (1+x)^n \tag{10.2}$$

  Of course, this is nothing new — it's what we started out with when we worked out the formula for $A(x,y)$. We just did this to become more familiar with the notation and manipulation.

- Now let's look at $[x^k]\, A(x,y)$. From (10.1) and the formula for a geometric series,

$$\begin{aligned} A(x,y) \;&=\; \frac{1}{(1-y)-xy} \;=\; \frac{1/(1-y)}{1-xy/(1-y)} \\ &=\; \sum_{k\ge 0} \frac{1}{1-y}\left(\frac{xy}{1-y}\right)^k \;=\; \sum_{k\ge 0} \frac{1}{1-y}\left(\frac{y}{1-y}\right)^k x^k. \end{aligned}$$

  Thus $[x^k]\, A(n,k) = \frac{1}{1-y}\left(\frac{y}{1-y}\right)^k$. In other words, we have the generating function

$$\sum_{n\ge 0} \binom{n}{k} y^n \;=\; \frac{y^k}{(1-y)^{k+1}}. \tag{10.3}$$

  This is new and we'll get more in a minute.

- We can replace the x and y in our generating functions by numbers. If we do that in (10.2) it's not very interesting. Let's do it in (10.3). We must be careful: The sum on the left side is infinite and so convergence is an issue. With $y = 1/3$ we have

$$\sum_{n\ge 0} \binom{n}{k} 3^{-n} \;=\; \frac{3^k}{2^{k+1}}, \tag{10.4}$$

  and it can be shown that the sum converges. So this is a new result. On the other hand, if we set $y = 2$ instead the series would have been $\sum \binom{n}{k} 2^n$ which diverges to infinity. The right side of (10.3) is not infinity but $(-1)^{k+1}2^k$, which is nonsensical for a sum of positive terms. That's a warning that something is amiss, namely a lack of convergence.

- Returning to (10.1), let's set $x = y$. In that case, we obtain

$$\sum_{n,k\ge 0} \binom{n}{k} x^{n+k} \;=\; A(x,x) \;=\; \frac{1}{1-x-x^2}. \tag{10.5}$$

  What is the coefficient of x^m on the left side? You should be able to see that it will be the sum of $\binom{n}{k}$ over all n and k such that $n+k = m$. Thus $n = m-k$ and so

$$\sum_{k\ge 0} \binom{m-k}{k} \;=\; [x^m]\left(\frac{1}{1-x-x^2}\right).$$

In the next section, we will see how to obtain such coefficients, which turn out to be the Fibonacci numbers. Convergence is not an issue: the sum on the left is finite since the binomial coefficients are nonzero only when $m - k \geq k$, that is $k \leq m/2$. □

There are two important differences in the study of generating functions here and in calculus. We've already noted one: convergence is usually not an issue as long as we know the coefficients make sense. The second is that our interest is in the reverse direction: We study generating functions to learn about their coefficients but in calculus one studies the coefficients to learn about the functions. For example, one might use the first few terms of the sum to estimate the value of the function.

The following simple theorem is important in combinatorial uses of generating functions. Some applications can be found in the exercises. It plays a crucial role in the Rule of Product in Section 10.4. Later, we will extend the theorem to generating functions with more than one variable.

Theorem 10.2 Convolution Formula *Let $A(x)$, $B(x)$, and $C(x)$ be generating functions. Then $C(x) = A(x)B(x)$ if and only if*

$$c_n = \sum_{k=0}^{n} a_k b_{n-k} \quad \text{for all } n \geq 0. \tag{10.6}$$

The sum can also be written $\sum_{k\geq 0} a_{n-k}b_k$ and also as the sum of $a_i b_j$ over all i, j such that $i + j = n$. We call (10.6) a convolution.

Proof: You should have no difficulty verifying that the two other forms given for the sum are in fact the same as $\sum a_k b_{n-k}$.

We first prove that $C(x) = A(x)B(x)$ gives the claimed summation. Since we are not concerning ourselves with convergence, we can multiply generating functions like polynomials:

$$A(x)B(x) = \left(\sum_{k\geq 0} a_k x^k\right)\left(\sum_{j\geq 0} b_j x^j\right) = \sum_{k,j\geq 0} a_k b_j x^{k+j} = \sum_{n\geq 0}\left(\sum_{k=0}^{n} a_k b_{n-k}\right)x^n,$$

where the last equality follows by letting $k+j = n$; that is, $j = n-k$. The sum on k stops at n because $j \geq 0$ is equivalent to $n - k \geq 0$, which is equivalent to $k \leq n$. This proves that $C(x) = A(x)B(x)$ implies (10.6).

Now suppose we are given (10.6). Multiply by x^n, sum over $n \geq 0$, let $j = n - k$ and reverse the steps in the previous paragraph to obtain

$$C(x) = \sum_{n\geq 0} c_n x^n = \sum_{k,j\geq 0} a_k b_j x^{k+j} = A(x)B(x).$$

We've omitted a few computational details that you should fill in. □

Here are a few generating functions that are useful to know about. The first you've already encountered, the second appears in Exercise 10.1.4, the third is an application of the convolution formula (Exercise 10.1.6), and the others are results from calculus.

$$\sum_{k=0}^{\infty}(ar^k)x^k = \frac{a}{1-rx}, \tag{10.7}$$

$$\sum_{k=0}^{\infty}\binom{r}{k}x^k = (1+x)^r \quad \text{where } \binom{r}{k} = \frac{r(r-1)\cdots(r-k+1)}{k!} \text{ for all } r, \tag{10.8}$$

$$\sum_{n=0}^{\infty}\left(\sum_{k=0}^{n} a_k\right)x^n = \frac{1}{1-x}\sum_{n\geq 0} a_n x^n, \tag{10.9}$$

$$\sum_{k=0}^{\infty} \frac{a^k x^k}{k!} \;=\; e^{ax}, \tag{10.10}$$

$$\sum_{k=1}^{\infty} \frac{a^k x^k}{k} \;=\; -\ln(1-ax). \tag{10.11}$$

Exercises

These exercises will give you some practice manipulating generating functions.

10.1.1. Let $p = 1 + x + x^2 + x^3$, $q = 1 + x + x^2 + x^3 + x^4$, and $r = \frac{1}{1-x}$.

(a) Find the coefficient of x^3 in p^2; in p^3; in p^4.

(b) Find the coefficient of x^3 in q^2; in q^3; in q^4.

(c) Find the coefficient of x^3 in r^2; in r^3; in r^4.

(d) Can you offer a simple explanation for the fact that p, q and r all gave the same answers?

(e) Repeat (a)–(c), this time finding the coefficient of x^4. Explain why some are equal and some are not.

10.1.2. Find the coefficient of x^2 in each of the following.

(a) $(2 + x + x^2)(1 + 2x + x^2)(1 + x + 2x^2)$

(b) $(2 + x + x^2)(1 + 2x + x^2)^2(1 + x + 2x^2)^3$

(c) $x(1 + x)^{43}(2 - x)^5$

10.1.3. Find the coefficient of x^{21} in $(x^2 + x^3 + x^4 + x^5 + x^6)^8$.
Hint. If you are clever, you can do this without a lot of calculation.

10.1.4. This exercise explores the general binomial theorem, geometric series and related topics. Part (a) requires calculus.

(a) Let r be any real number. Use Taylor's Theorem without worrying about convergence to prove

$$(1+z)^r \;=\; \sum_{k\geq 0} \binom{r}{k} z^k \quad \text{where} \quad \binom{r}{k} \;=\; \frac{r(r-1)\cdots(r-k+1)}{k!}.$$

If you're familiar with some form of Taylor's Theorem with remainder, use it to show that, for some $C > 0$, the infinite sum converges when $|z| < C$. (The largest possible value is $C = 1$, but you may find it easier to use a smaller value.)

(b) Use the previous result to obtain the geometric series formula: $\sum_{k\geq 0} az^k = a/(1-z)$.

(c) Show that $\sum_{k=0}^{n} az^k \;=\; (a - az^{n+1})/(1-z)$.

(d) Find a *simple* formula for the coefficient of x^n in $(1-ax)^{-2}$.

10.1.5. In this exercise we'll explore the effect of derivatives. Let $A(x) = \sum_{m=0}^{\infty} a_m x^m$, the ordinary generating function for the sequence a. In each case, first answer the question for $k = 1$ and $k = 2$ and then for general k.

(a) What is $[x^n]\ (x^k A(x))$, that is, the coefficient of x^n in $x^k A(x)$?

(b) Show that $[x^n] \left(\frac{d}{dx}\right)^k A(x) \;=\; \frac{(n+k)!\, a_{n+k}}{n!}$. This notation means compute the kth derivative of $A(x)$ and then find the coefficient of x^n in the generating function. It can also be written $[x^n]\, A^{(k)}(x)$.

(c) Show that $[x^n] \left(x\frac{d}{dx}\right)^k A(x) \;=\; n^k a_n$. This notation means that you repeat alternately the operations of differentiating and multiplying by x a total of k times each. For example, when $k = 2$ we have $x(xA'(x))'$.

10.1.6. Using Theorem 10.2 or otherwise, do the following.

(a) Prove: If $c_n = a_0 + a_1 + \cdots + a_n$, then $C(x) = A(x)/(1-x)$.

(b) Simplify $\binom{n}{0} - \binom{n}{1} + \cdots + (-1)^k\binom{n}{k}$ when $n > 0$.

(c) Suppose that d_n is the sum of $a_i b_j c_k$ over all $i, j, k \geq 0$ such that $i + j + k = n$. Express $D(x)$ in terms of $A(x)$, $B(x)$, and $C(x)$.

10.1.7. Suppose that $|r| < 1$. Obtain a formula for $\sum_{n\geq 0}\binom{n}{k}r^n$ as a function of k and r. Show that the sum converges by using the ratio test for series.

10.1.8. Note that $(1+x)^{m+n} = (1+x)^m(1+x)^n$. Note that the coefficients of powers of x in $(1+x)^{m+n}$, $(1+x)^m$, and $(1+x)^n$ are binomial coefficients. Use Theorem 10.2 to prove *Vandermonde's formula*:

$$\binom{m+n}{k} = \sum_{i=0}^{k}\binom{m}{i}\binom{n}{k-i}.$$

This is one of the many identities that are known for binomial coefficients.
Hint. Remember that n and k in (10.6) can be replaced by other variables. Look at the index and limits on the summation.

10.1.9. Find a *simple* expression for $\sum_i (-1)^i\binom{m}{i}\binom{m}{k-i}$, where the sum is over all values of i for which the binomial coefficients in the sum are defined.

10.1.10. The results given here are referred to as *bisection of series*. Let $A(x) = \sum_{n=0}^{\infty} a_n x^n$.

(a) Show that $(A(x) + A(-x))/2$ is the generating function for the sequence b_n which is zero for odd n and equals a_n for even n.

(b) What is the generating function for the sequence c_n which is zero for even n and equals a_n for odd n?

(c) Evaluate $\sum_{k\geq 0}\binom{n}{2k}x^{2k}$ where x is a real number. In particular, what is $\sum_{k\geq 0}\binom{n}{2k}$?

*10.1.11. Fix $k > 1$ and $0 \leq j < k$. If you are familiar with kth roots of unity, generalize the Exercise 10.1.10 to the sequence b_n which is a_n when $n + j$ is a multiple of k and is zero otherwise:

$$B(x) = \frac{1}{k}\sum_{s=0}^{k-1}\omega^{js}A(\omega^s x),$$

where $\omega = \exp(2\pi i/k)$, a primitive kth root of unity. (The result is called *multisection of series*.)

10.1.12. Evaluate $s_k = \displaystyle\sum_{n=0}^{\infty}\binom{2n}{k}2^{-n}$.

*10.1.13. Using Exercise 10.1.11, show that

$$\sum_{n=0}^{\infty}\frac{x^{3n}}{(3n)!} = \frac{e^x}{3} + \frac{2\cos(x\sqrt{3}/2)}{3e^{x/2}}$$

and develop similar formulas for $\sum p^{3n+1}/(3n+1)!$ and $\sum p^{3n+2}/(3n+2)!$.

*10.1.14. We use the terminology from the Principle of Inclusion and Exclusion (Theorem 4.1 (p. 95)). Also, let E_k be the number of elements of S that lie in exactly k of the sets $S_1, S_2, \ldots, S_m$.

(a) Using the Rules of Sum and Product (not Theorem 4.1), prove that

$$N_r = \sum_{k\geq 0} \binom{r+k}{r} E_{r+k}.$$

(b) If the generating functions corresponding to $E_0, E_1, \ldots$ and $N_0, N_1, \ldots$ are $E(x)$ and $N(x)$, conclude that $N(x) = E(x+1)$.

(c) Use this to conclude that $E(x) = N(x-1)$ and then deduce the extension of the Principle of Inclusion and Exclusion:

$$E_k = \sum_{i\geq 0} (-1)^i \binom{k+i}{i} N_{k+i}.$$

10.2 Solving a Single Recursion

In this section we'll use ordinary generating functions to solve some simple recursions, including two that we were unable to solve previously: the Fibonacci numbers and the number of unlabeled full binary RP-trees.

Example 10.2 Fibonacci numbers Let F_n be the number of n long sequences of zeroes and ones with no consecutive ones. We can easily see that $F_1 = 2$ and $F_2 = 3$, but what is the general formula?

Suppose that $t_1, \ldots, t_n$ is an arbitrary sequence of desired form. We want to see what happens when we remove the end of the sequence, so we assume that $n > 1$. If $t_n = 0$, then $t_1, \ldots, t_{n-1}$ is also an arbitrary sequence of the desired form. Now suppose that $t_n = 1$. Then $t_{n-1} = 0$ and so, if $n > 2$, $t_1, \ldots, t_{n-2}$ is an arbitrary sequence of the desired form. All this is reversible: Suppose that $n > 2$. The following two operations produce all n long sequences of the desired form *exactly once*.

- Let $t_1, \ldots, t_{n-1}$ be an arbitrary sequence of the desired form. Set $t_n = 0$.
- Let $t_1, \ldots, t_{n-2}$ be an arbitrary sequence of the desired form. Set $t_{n-1} = 0$ and $t_n = 1$.

Since all n long sequences of the desired form are obtained exactly once this way, the Rule of Sum yields the recursion

$$F_n = F_{n-1} + F_{n-2} \quad \text{for} \quad n > 2. \tag{10.12}$$

Here are the first few values.

n	0	1	2	3	4	5	6	7	8	9	10
F_n	1	2	3	5	8	13	21	34	55	89	144

These numbers, called the *Fibonacci numbers*, were studied in Exercise 1.4.10, but we couldn't solve the recursion there. Now we will.

First, we want to adjust (10.12) so that it holds for all $n \geq 0$. To do this we define F_n when n is small and introduce a new sequence c_n to "correct" the recursion for small n;

$$F_n = F_{n-1} + F_{n-2} + c_n, \tag{10.13}$$

where $F_0 = 1$, $F_k = 0$ for $k < 0$, $c_0 = c_1 = 1$, and $c_n = 0$ for $n \geq 2$. This recursion is now valid for $n \geq 0$. Let $F(x)$ be the generating function for $F_0, F_1, \ldots$. In the following series of equations, steps without explanation require only simple algebra.

$$\begin{aligned}
F(x) &= \sum_{n=0}^{\infty} F_n x^n && \text{by defintion} \\
&= \sum_{n=0}^{\infty} (F_{n-1} + F_{n-2} + c_n) x^n && \text{by (10.13)} \\
&= \sum_{n=0}^{\infty} \left(x F_{n-1} x^{n-1} + x^2 F_{n-2} x^{n-2} + c_n x^n \right) \\
&= x \sum_{n=0}^{\infty} F_{n-1} x^{n-1} + x^2 \sum_{n=0}^{\infty} F_{n-2} x^{n-2} + \sum_{n=0}^{\infty} a_n x^n \\
&= x \sum_{i=1}^{\infty} F_i x^i + x^2 \sum_{k=0}^{\infty} F_k x^k + 1 + x && \text{by definition} \\
&= x F(x) + x^2 F(x) + 1 + x.
\end{aligned}$$

In summary, $F(x) = 1 + x + (x + x^2)F(x)$. We can easily solve this equation:

$$F(x) = \frac{1+x}{1-x-x^2}. \tag{10.14}$$

Now what? We want to find a formula for the coefficient of x^n in $F(x)$. We could try using Taylor's Theorem. Unfortunately, $F^{(n)}(x)$ appears to be extremely messy. What alternative do we have?

Remember partial fractions from calculus? If not, you should read Appendix D (p. 387). Using partial fractions, we will be able to write $F(x) = A/(1-ax) + B/(1-bx)$ for some constants a, b, A and B. Since the formula for summing geometric series is $1 + ax + (ax)^2 + \cdots = 1/(1-ax)$, we will have $F_n = Aa^n + Bb^n$. There is one somewhat sneaky point here. We want to factor a polynomial of the form $1 + cx + dx^2$ into $(1-ax)(1-bx)$. To do this, let $y = 1/x$ and multiply by y^2. The result is $y^2 + cy + d = (y-a)(y-b)$. Thus a and b are just the roots of $y^2 + cy + d = 0$. In our case we have $y^2 - y - 1 = 0$.

Let's carry out the partial fraction approach. We have

$$1 - x - x^2 = (1-ax)(1-bx) \qquad \text{where} \quad a,\ b = \frac{1 \pm \sqrt{5}}{2}.$$

(Work it out.) For definitiveness, let a be associated with the $+$ and b with the $-$. To get some idea of the numbers we are working with, $a = 1.618\cdots$ and $b = -.618\cdots$. By expanding in partial fractions, you should be able to derive

$$F(x) = \frac{1+x}{1-x-x^2} = \frac{1+a}{\sqrt{5}(1-ax)} - \frac{1+b}{\sqrt{5}(1-bx)}.$$

Now use geometric series and the algebraic observations $1 + a = a^2$ and $1 + b = b^2$ to get

$$F_n = \frac{a^{n+2}}{\sqrt{5}} - \frac{b^{n+2}}{\sqrt{5}}. \tag{10.15}$$

It is not obvious that this expression is even an integer, much less equal to F_n. If you're not convinced, you might like to calculate a few values.

Since $|b| < 1$, $\left| b^{n+2}/\sqrt{5} \right| < 1/\sqrt{5} < 1/2$. Thus we have the further observation that F_n is the integer closest to $a^{n+2}/\sqrt{5} = (1.618\cdots)^{n+2}/2.236\cdots$. For example $a^4/\sqrt{5} = 3.065\cdots$ which is close to $F_2 = 3$ and $a^{12}/\sqrt{5} = 144.001\cdots$, which is quite close to $F_{10} = 144$. Of course, the approximations get better as n gets larger since the error is bounded by a large power of b and $|b| < 1$. □

The method that we have just used works for many other recursions, so it is useful to lay it out as a series of steps. Although our description is for a singly indexed recursion, it can be applied to the multiply indexed case as well.

A procedure for solving recursions

Here is a six step procedure for solving recursions. It is not guaranteed to work because it may not be possible to carry out all of the steps. Let the sequence be a_n.

1. *Adjust the recursion so that it is valid for all n. In particular, a_n should be defined for all n and $a_n = 0$ for $n < 0$. You may need to introduce a "correcting" sequence c_n as in (10.13).*
2. *Introduce the generating function $A(x) = \sum_{n\geq 0} a_n x^n$.*
3. *Substitute the recursion into the summation for $A(x)$.*
4. *Rearrange the result so that you can recognize other occurrences of $A(x)$ and so get rid of summations. (This is not always possible; it depends on what the recursion is like.)*
5. *If possible, solve the resulting equation to obtain an explicit formula for $A(x)$.*
6. *By partial fractions, Taylor's Theorem or whatever, obtain an expression for the coefficient of x^n in this explicit formula for $A(x)$. This is a_n.*

You should go back to the previous example and find out where each step was done.

Example 10.3 Fibonacci numbers continued

Setting $y = x$ in (10.1) gives $1/(1-x-x^2)$, which we'll call $H(x)$. This is nearly $F(x) = (1+x)/(1-x-x^2)$ of the previous example, suggesting that there is a connection between binomial coefficients and Fibonacci numbers. Let's explore this.

Writing $F(x)/(1+x) = H(x)$ is not a good idea since the coefficient of x^n on the left side is $F_n - F_{n-1} + F_{n-2} - \cdots$ and we'd like to find a simpler connection if we can. Writing the equation as $(1+x)H(x) = F(x)$ is better since the coefficient of x^n on the left side is just $h_n + h_{n-1}$.

It would be even better if we could avoid the factor of $(1+x)$ and have a monomial instead, since then we would not have to add two terms together. You might like to try to find something like that. After some work, we found that $1 + xF(x) = H(x)$, which is easily verified by using the formulas for $H(x)$ and $F(x)$. You should convince yourself that for $n > 0$ the coefficient of x^n on the left side is F_{n-1} and so $F_n = h_{n+1}$. In fact, some people call $1, 1, 2, 3, \ldots$ the Fibonacci numbers and then h_n is the nth Fibonacci number and $1/(1-x-x^2)$ is the generating function for the Fibonacci numbers. Still others call $0, 1, 1, 2, 3, \ldots$ the Fibonacci numbers and then $x/(1-x-x^2)$ is the generating function for them. Anyway, with $a_{j,i} = \binom{j}{i}$, *our* Fibonacci number F_n is the coefficient of x^{n+1} in $H(x)$. By (10.1),

$$H(x) \;=\; \sum_{j=0}^{\infty}\sum_{i=0}^{\infty}\binom{j}{i}x^i x^j.$$

Note that the coefficient of x^{n+1} on the right side is the sum of $\binom{j}{i}$ over all nonnegative i and j such that $i + j = n + 1$. Hence $F_n = \sum_{i=0}^{n+1}\binom{n+1-i}{i}$. This is such a simple expression that it should have a direct proof. We leave that as an exercise. □

Example 10.4 The worst case time for merge sorting Let $M(n)$ be the maximum number of comparisons needed to merge sort a list of n items. (Merge sorting was discussed in Example 7.13 and elsewhere.) The best way to do a merge sort is to split the list as evenly as possible. If n is even, we can divide the list exactly in half. It takes at most $M(n/2)$ comparisons to merge sort each of the two halves and then at most $n-1 < n$ comparisons to merge the two resulting lists. Thus $M(n) < n + 2M(n/2)$. We'd like to use this to define a recursion, but there's a problem: $n/2$ may not be even.

How can we avoid this? We can just look at those values of n which are powers of 2. For example, the fact that $M(1) = 0$ gives us

$$\begin{aligned} M(8) &< 8 + 2M(4) < 8 + 2\big(4 + 2M(2)\big) \\ &< 8 + 2\Big(4 + 2\big(2 + 2M(1)\big)\Big) = 8 + 2(4+4) = 24. \end{aligned}$$

How can we set up a recursion that only looks at values of n which are a power of 2? We let $m_k = M(2^k)$. Then

$$m_0 = M(1) = 0 \quad \text{and} \quad m_k = M(2^k) < 2^k + 2M(2^{k-1}) = 2^k + 2m_{k-1}.$$

So far we have only talked about solving recursive relations that involve equality, but this is an inequality. What can we do about that?

If we define c_k by

$$c_0 = 0 \quad \text{and} \quad c_k = 2^k + 2c_{k-1} \text{ for } k > 0, \tag{10.16}$$

then it follows that $m_k \le c_k$. We'll solve (10.16) and so get a bound for $m_k = M(2^k)$.

Before calculating the general solution, it may be useful to use the recursion to calculate a few values. This might lead us to guess what the solution is. Even if we can't guess the solution, we'll have some special cases of the general solution available so that we'll be able to partially check the general solution when we finally get it. It's a good idea to get in the habit of making such checks because it is *very easy* to make algebra errors when manipulating generating functions.

From (10.16), the first few values of c_k are

$$c_0 = 0, \quad c_1 = 2, \quad c_2 = 2 \cdot 2^2 = 2^3, \quad c_3 = 3 \cdot 2^3 \quad \text{and} \quad c_4 = 4 \cdot 2^4.$$

This strongly suggests that $c_k = k2^k$. You should verify that this is correct by using (10.16) and induction.

Since we have the answer, why bother with generating functions? We want to study generating function techniques so that you can use them in situations where you can't guess the answer. This problem is a rather simple one, so the algebra won't obscure the use of the techniques.

For Step 1, rewrite (10.16) as

$$c_k = 2^k + 2c_{k-1} + a_k \quad \text{for} \quad k \ge 0,$$

where $c_k = 0$ for $k < 0$, $a_0 = -1$, and $a_n = 0$ for $n > 0$. Now

$$\begin{aligned} C(x) &= \sum_{k=0}^{\infty} c_k x^k && \text{This is Step 2.} \\ &= \sum_{k=0}^{\infty} (2^k + 2c_{k-1} + a_k)x^k && \text{This is Step 3.} \\ &= \sum_{k=0}^{\infty} (2x)^k 2x \sum_{k=0}^{\infty} c_{k-1}x^{k-1} - 1 \\ &= \frac{1}{1-2x} + 2xC(x) - 1. && \text{This is Step 4.} \end{aligned}$$

For Step 5 we have $C(x) = 2x/(1-2x)^2$. Partial fractions (Step 6) leads to

$$C(x) \;=\; \frac{1}{(1-2x)^2} - \frac{1}{1-2x} \;=\; \sum \binom{-2}{k}(-2x)^k - \sum (2x)^k.$$

Thus $c_k = 2^k\left((-1)^k\binom{-2}{k} - 1\right) = k2^k$. Hence $M(n) \le n\log_2 n$ when n is a power of 2. How good is this bound? What happens when n is not a power of 2? It turns out that $n\log_2 n$ is a fairly good estimate for $M(n)$ for all n, but we won't prove it. □

Perhaps you've noticed that when we obtain a rational function (i.e., a quotient of two polynomials) as a generating function, the denominator is, in some sense, the important part. We can state this more precisely: For rational generating functions, the recursion determines the denominator and the initial conditions interacting with the recursion determine the numerator. No proof of this claim will be given here. A related observation is that, if we have the same denominators for two rational generating functions $A(x)$ and $B(x)$ that have been reduced to lowest terms, then the coefficients a_n and b_n have roughly the same rate of growth for large n; i.e., we usually have $a_n = \Theta(b_n)$.*

Example 10.5 Counting unlabeled full binary RP-trees Let b_n be the number of unlabeled full binary RP-trees with n leaves. By Example 9.4 (p. 251), the number of such trees is the Catalan number C_{n-1}. See Example 1.13 (p. 15) for more examples of things that are counted by the Catalan numbers.

The recursion

$$b_n \;=\; \sum_{k=1}^{n-1} b_k b_{n-k} \quad \text{if } n > 1 \tag{10.17}$$

with $b_1 = 1$ was derived as (9.3). Recall that $b_1 = 1$ and b_0 was not defined. Let's use our procedure to find b_n. Here it is, step by step.

1. Since (10.17) is nearly a convolution, we define $b_0 = 0$ to make it a convolution:

$$b_n \;=\; \sum_{k=0}^{n} b_k b_{n-k} + a_n,$$

   where $a_1 = 1$ and $a_n = 0$ for $n \ne 1$.
2. Let $B(x) = \sum_{n\ge 0} b_n x^n$.
3. $B(x) = \sum_{n\ge 0}\sum_{k=0}^{n} b_k b_{n-k} x^n + x$.
4. By the formula for convolutions, we now have

$$B(x) \;=\; B(x)B(x) + x. \tag{10.18}$$

5. The quadratic equation $B = x + B^2$ has the solution $B = (1 \pm \sqrt{1-4x})/2$. Since $B(0) = b_0 = 0$, the minus sign is the correct choice. Thus

$$B(x) \;=\; \frac{1-\sqrt{1-4x}}{2}.$$

6. By Exercise 10.1.4,

$$(1+z)^r \;=\; \sum_{n=0}^{\infty} \binom{r}{n} z^n, \quad \text{where } \binom{r}{n} = \frac{r(r-1)\cdots(r-n+1)}{n!}.$$

* This notation is discussed in Appendix B. It means there exist positive constants A and B such that $Aa_n \le b_n \le Ba_n$.

Now for some algebra. With $n > 0$, $r = 1/2$ and $z = -4x$ we obtain

$$\begin{aligned} b_n &= -\frac{1}{2}\binom{\frac{1}{2}}{n}(-4)^n \\ &= \left(\frac{1}{2}(2^n)\right)\left(\frac{\frac{1}{2}(\frac{1}{2}-1)(\frac{1}{2}-2)\cdots(\frac{1}{2}-n+1)}{n!}2(-2)^{n-1}\right) \\ &= \left(\frac{2^{n-1}(n-1)!}{(n-1)!}\right)\left(\frac{(-1+2)(-1+4)\cdots(-1+2n-2)}{n!}\right) \\ &= \left(\frac{2\cdot 4\cdots(2n-2)}{(n-1)!}\right)\left(\frac{1\cdot 3\cdots(2n-3)}{n!}\right) \\ &= \frac{(2n-2)!}{(n-1)!n!} = \frac{1}{n}\binom{2n-2}{n-1}. \end{aligned}$$

As remarked at the beginning of the example, this number is the Catalan number C_{n-1}. Thus $C_n = \frac{1}{n+1}\binom{2n}{n}$. □

Exercises

10.2.1. Solve the following recursions by using generating functions.

(a) $a_0 = 0$, $a_1 = 1$ and $a_n = 5a_{n-1} - 6a_{n-2}$ for $n > 1$.

(b) $a_0 = a_1 = 1$ and $a_{n+1} = a_n + 6a_{n-1}$ for $n > 0$.

(c) $a_0 = 0$, $a_1 = a_2 = 1$ and $a_n = a_{n-1} + a_{n-2} + 2a_{n-3}$ for $n > 2$.

(d) $a_0 = 0$ and $a_n = 2a_{n-1} + n$ for $n > 0$.

10.2.2. Let $S(n)$ be the number of moves needed to solve the Towers of Hanoi puzzle. In Exercise 7.3.9 you were asked to show that $S(1) = 1$ and $S(n) = 2S(n-1) + 1$ for $n > 1$.

(a) Use this recursion to obtain the generating function for S.

(b) Use the generating function to determine $S(n)$.

10.2.3. Show without generating functions that $\binom{n+1-i}{i}$ is the number of n long sequences of zeroes and ones with exactly i ones, none of them adjacent. Use this result to prove the formula $F_n = \sum_{i\geq 0}\binom{n+1-i}{i}$ that was derived in the Example 10.3 via generating functions.

10.2.4. Let s_n be the number of n long sequences of zeroes, ones and twos with no adjacent ones and no adjacent twos. Let $s_0 = 1$; i.e., there is one empty sequence.

(a) Let k be the position of the last zero in such a sequence. If there is no zero, set $k = 0$. Show that the last $n - k$ elements in the sequence consist of an alternating pattern of ones and twos and that the only restriction on the first $k - 1$ elements in the sequence is that there be no adjacent ones and no adjacent twos.

(b) By considering all possibilities for k in (a), conclude that, for $n > 0$,

$$s_n = 2 + 2s_0 + 2s_1 + \cdots + 2s_{n-2} + s_{n-1}.$$

(c) Use the convolution formula to deduce

$$S(x) = (1 + 2x + 2x^2 + 2x^3 + \cdots)(1 + s_0x + s_1x^2 + s_2x^3 + \cdots) = \left(1 + \frac{2x}{1-x}\right)(1 + xS(x)).$$

(d) Conclude that $S(x) = (1 + x)/(1 - 2x - x^2)$.

(e) Find a formula for s_n and check it for $n = 0, 1, 2$.

(f) Show that s_n is the integer closest to $(1 + \sqrt{2})^{n+1}/2$.

10.2.5. The usual method for multiplying two polynomials of degree $n-1$, say

$$P_1(x) = a_{0,1} + a_{1,1}x + \cdots + a_{n-1,1}x^{n-1} \quad \text{and} \quad P_2(x) = a_{0,2} + a_{1,2}x + \cdots + a_{n-1,2}x^{n-1}$$

requires n^2 multiplications to form the products $a_{i,1}a_{j,2}$ for $0 \le i, j < n$. These are added together in the appropriate way to form the $2n-1$ sums that constitute the coefficients of the product $P_1(x)P_2(x)$. There is a less direct method that requires less multiplications. For simplicity, suppose that $n = 2m$.

- First, split the polynomials in "half": $P_i(x) = L_i(x) + x^m H_i(x)$, where L_i and H_i have degree at most $m-1$.
- Second, let $A = H_1H_2$, $B = L_1L_2$ and $C = (H_1 + L_1)(H_2 + L_2)$.
- Third, note that $P_1P_2 = Ax^{2m} + B + (C - A - B)x^m$.

(a) Prove that the formula for P_1P_2 is correct.

(b) Let $M(n)$ be the least number of multiplications we need in a general purpose algorithm for multiplying two polynomials of degree $n-1$. show that $M(2m) \le 3M(m)$.

(c) Use the previous result to derive an upper bound for $M(n)$ when n is a power of 2 that is better than n^2. (Your answer should be $M(n) \le n^c$ where $c = 1.58\cdots$.) How does this bound compare with n^2 when $n = 2^{10} = 1024$?

Your bound will give a bound for all n since, if $n \le 2^k$, we can fill the polynomials out to degree 2^k by introducing high degree terms with zero coefficients. This gives $M(n) \le M(2^k)$.

(d) Show how the method used to obtain the bound multiplies $1 + 2x - x^2 + 3x^3$ and $5 + 2x - x^3$.

*(e) It may be objected that our method could lead to such a large number of additions and subtractions that the savings in multiplication may be lost. Does this happen? Justify your answer.

10.2.6. Let t_n be the number of n-vertex unlabeled binary RP-trees. (Each vertex has 0, 1 or 2 children.)

(a) Derive the recursion

$$t_1 = 1 \quad \text{and} \quad t_{n+1} = t_n + \sum_{k=1}^{n-1} t_k t_{n-k} \quad \text{for} \quad n > 0.$$

(b) With $t_0 = 0$, derive an equation for the generating function $T(x) = \sum_{n\ge 0} t_n x^n$.

(c) Solve the equation in (b) to obtain

$$T(x) = \frac{1 - x - \sqrt{1 - 2x - 3x^2}}{2x}$$

and explain the choice of sign before the square root.

10.2.7. Let $c_1, \ldots, c_k$ be arbitrary real numbers. If you are familiar with partial fractions, explain why the solution to the recursion $a_n = c_1 a_{n-1} + \cdots + c_k a_{k-k}$ has the form $a_n = \sum_{i=1}^{m} P_i(n) r_i^n$ for all sufficiently large n, where $P_i(n)$ is a polynomial of degree less than d_i, the r_i are all different, and

$$1 - c_1 x - \cdots - c_k x^k = \prod_{i=1}^{m} (1 - r_i x)^{d_i}.$$

How can the polynomials $P_n(n)$ be found without using partial fractions?

10.3 Manipulating Generating Functions

Almost anything we do with generating functions can be regarded as manipulation, so what does the title of this section refer to? We mean the use of tools from algebra and calculus to obtain information from generating functions. We've already seen some examples of one tool being used: partial fractions. In this section we'll focus on two others; (i) the manipulation of generating functions to obtain, when possible, simple recursions and (ii) the interplay of derivatives with generating functions. Some familiarity with calculus is required. The results in this section are used some in later sections, but they are not essential for understanding the concepts introduced there.

Obtaining Recursions

Suppose we have an equation that determines a generating function $B(x)$; for example, $B(x) = \frac{1-\sqrt{1-4x}}{2}$. The basic idea for obtaining a recursion for $B(x)$ is to rewrite the equation so that $B(x)$ appears in expressions that are simple and so that the remaining expressions are easy to expand in power series. Once a simple form has been found, equate coefficients of x^n on the two sides of the equation. We'll explore this idea here.

Example 10.6 Rational functions and recursions Suppose that $B(x) = P(x)/Q(x)$ where $P(x)$ and $Q(x)$ are polynomials. Expressions that involve division are usually not easy to expand unless the divisor is a product of linear factors with integer coefficients. Thus, we would usually rewrite our equation as $Q(x)B(x) = P(x)$ and then equate coefficients. This gives us a recursion for the b_i's which is linear and has constant coefficients.

The description of the procedure is a bit vague, so let's look at an example. When we study systems of recursions in the next chapter, we will show that the number of ways to place nonoverlapping dominoes on a 2 by n board has the generating function

$$C(x) \;=\; \frac{1-x}{1-3x-x^2+x^3}.$$

Thus $P(x) = 1-x$ and $Q(x) = 1-3x-x^2+x^3$. Using our plan, we have

$$(1-3x-x^2+x^3)C(x) \;=\; 1-x. \tag{10.19}$$

There are now various ways we can proceed:

Keep all subscripts nonnegative: When $n \geq 3$, the coefficient of x^n on the right side is 0 and the coefficient on the left side is $c_n - 3c_{n-1} - c_{n-2} + c_{n-3}$, so all the subscripts are nonnegative. Rearranging this,

$$c_n \;=\; 3c_{n-1} + c_{n-2} - c_{n-3} \quad \text{for } n \geq 3.$$

The values of a_0, a_1 and a_2 are given by initial conditions. Looking at the coefficients of x^0, x^1 and x^2 on both sides of (10.19), we have

$$a_0 \;=\; 1 \qquad a_1 - 3a_0 \;=\; -1 \qquad a_2 - 3a_1 - a_0 \;=\; 0.$$

Solving we have $a_0 = 1$, $a_1 = 2$ and $a_2 = 7$. (You might want to try deriving the recursion directly. It's not easy, but it's not an unreasonable problem for you at this time.)

Allow negative subscripts: We now allow negative subscripts, with the understanding that $a_n = 0$ if $n < 0$. Proceeding as above, we get $c_n - 3c_{n-1} - c_{n-2} + c_{n-3} = 0$ provided $n \geq 2$. Thus we get the same recursion, but now $n \geq 2$ and the initial conditions are only $a_0 = 1$ and $a_1 = 2$ since a_3 is given by the recursion.

Avoid initial conditions: Now we not only allow negative subscripts, we also do not restrict n. From (10.19) we have

$$c_n - 3c_{n-1} - c_{n-2} + c_{n-3} = b_n, \quad \text{where } b_n = [x^n]\,(1-x).$$

Thus we have the recursion

$$c_n \;=\; 3c_{n-1} + c_{n-2} - c_{n-3} + b_n \quad \text{for } n \geq 0,$$

where $b_0 = 1$, $b_1 = -1$ and $b_n = 0$ otherwise. □

The ideas are not limited to ratios of polynomials, but then it's not always clear how to proceed. In the next example, we use the fact that e^{-x} has a simple power series.

Example 10.7 Derangements In the next chapter, we obtain, as (11.17) the formula

$$D(x) \;=\; \sum_{n=0}^{\infty} D_n x^n/n! \;=\; \frac{e^{-x}}{1-x}; \tag{10.20}$$

in other words, $e^{-x}/(1-x)$ is the ordinary generating function for the numbers $d_n = D_n/n!$. We can get rid of fractions in (10.20) by multiplying by $(1-x)$. Since

$$e^{-x} \;=\; \sum_{n=0}^{\infty} \frac{(-1)^n x^n}{n!},$$

equating coefficients of x^n on both sides of $(1-x)D(x) = e^{-x}$ gives us

$$\frac{D_n}{n!} - \frac{D_{n-1}}{(n-1)!} \;=\; \frac{(-1)^n}{n!}.$$

Rearranging leads to the recursion $D_n = nD_{n-1} + (-1)^n$. A direct combinatorial proof of this recursion is known, but it is difficult. □

One method for solving a differential equation is to write the unknown function as a power series $y(x) = \sum a_n x^n$, use the differential equation to obtain a recursion for the a_n, and finally use the recursion to obtain information about the a_n's and hence $y(x)$. Here we proceed differently. Sometimes a recursion may lead to a differential equation which can solved to obtain the generating function. Sometimes a differential equation can be found for a known generating function and then be used to obtain a recursion. We consider the latter approach in the next example. What sort of differential equation should we look for? Linear equations with polynomial coefficients give the simplest recursions.

Example 10.8 A recursion for unlabeled full binary RP-trees In Example 10.5 we found that the generating function for unlabeled full binary RP-trees is $B(x) = \frac{1-\sqrt{1-4x}}{2}$. We then obtained an explicit formula for b_n by expanding $\sqrt{1-4x}$ in a power series. Instead, we could obtain a differential equation which would lead to a recursion.

We can proceed in various ways to obtain a simple differential equation. One is to observe that $2B(x) - 1 = -(1-4x)^{1/2}$ and differentiate both sides to obtain $2B'(x) = 2(1-4x)^{-1/2}$. Multiply by $1-4x$:

$$2(1-4x)B'(x) = 2(1-4x)^{1/2} \;=\; -(2B(x)-1).$$

Thus $2B'(x) - 8xB'(x) + 2B(x) = 1$. Replacing $B(x)$ by its power series we obtain

$$\sum 2nb_n x^{n-1} - \sum 8nb_n x^n + \sum 2b_n x^n \;=\; 1.$$

Replacing the first sum by $\sum 2(n+1)b_{n+1}x^n$ and equating coefficients of x^n gives

$$2(n+1)b_{n+1} - 8nb_n + 2b_n \;=\; 0 \quad \text{for} \quad n > 0.$$

After some rearrangement, $b_{n+1} = (4n-2)b_n/(n+1)$ for $n > 0$. We already know that $b_1 = 1$, so we have the initial condition for the recursion. This recursion was obtained in Exercise 9.3.13 (p. 266) by a counting argument. □

Derivatives, Averages and Probability

The fact that $xA'(x) = \sum na_n x^n$ can be quite useful in obtaining information about averages. We'll explain how this works and then look at some examples.

Let $\mathcal{A}_n$ be a set of objects of size n; for example, some kind of n-long sequences or some kind of n-vertex trees. For each n, make $\mathcal{A}_n$ into a probability space using the uniform distribution:

$$\Pr(\alpha) = \frac{1}{|\mathcal{A}_n|} \text{ for all } \alpha \in \mathcal{A}_n.$$

(Probability is discussed in Appendix C.) Suppose that for each n we have a random variable X_n on $\mathcal{A}_n$ that counts something; for example, the number of ones in a sequence or the number of leaves on a tree. The average value (average number of ones or average number of leaves) is then $\mathbf{E}(X_n)$.

Now let's look at this in generating function terms. Let $a_{n,k}$ be the number of $\alpha \in \mathcal{A}_n$ with $X_n(\alpha) = k$; for example, the number of n-long sequences with k ones or the number of n-vertex trees with k leaves. Let $A(x,y)$ be the generating function $\sum_{n,k} a_{n,k}x^n y^k$. By the definition of expectation and simple algebra,

$$\mathbf{E}(X_n) \;=\; \sum_k k \Pr(X_n = k) \;=\; \sum_k k\frac{a_{n,k}}{|\mathcal{A}_n|} \;=\; \frac{\sum_k ka_{n,k}}{|\mathcal{A}_n|} \;=\; \frac{\sum_k ka_{n,k}}{\sum_k a_{n,k}}.$$

Let's look at the two sums in the last fraction.

$$\text{Since } [x^n]\,A(x,y) = \textstyle\sum_k a_{n,k}y^k, \quad \sum_k a_{n,k} = [x^n]A(x,1).$$

$$\text{Since } [x^n]\,\tfrac{\partial A(x,y)}{\partial y} = \textstyle\sum_k ka_{n,k}y^{k-1}, \quad \sum_k ka_{n,k} = [x^n]A_y(x,1),$$

where A_y stands for $\partial A/\partial y$. Putting this all together,

$$\mathbf{E}(X_n) \;=\; \frac{[x^n]\,A_y(x,1)}{[x^n]\,A(x,1)}. \tag{10.21}$$

We can use the same idea to compute variance. Recall that $\mathbf{var}(X_n) = \mathbf{E}(X_n^2) - \mathbf{E}(X_n)^2$. Since (10.21) tells us how to compute $\mathbf{E}(X_n)$, all we need is a formula for $\mathbf{E}(X_n^2)$. This is just like the

previous derivation except we need factors of k^2 multiplying $a_{n,k}$. We can get this by differentiating twice:

$$\sum_k k^2 a_{n,k} \;=\; [x^n]\,\frac{\partial(yA_y(x,y))}{\partial y}\bigg|_{y=1} \;=\; [x^n](A_{yy}(x,1)+A_y(x,1)). \tag{10.22}$$

This discussion has all been rather abstract. Let's apply it.

Example 10.9 Fibonacci sequences What is the average number of ones in an n long sequence of zeroes and ones containing no adjacent ones? We studied these sequences in Example 10.2 (p. 275), where we used the notation F_n. To be more in keeping with the previous discussion, let Let $f_{n,k}$ be the number of n long sequences containing exactly k ones. We need $F(x,y)=\sum_{n,k} f_{n,k}x^n y^k$.

In Example 10.16 we'll see how to compute $F(x,y)$ quickly, but for now the only tool we have is recursions, so it will take a bit longer. You should be able to extend the argument used to derive the recursion (10.12) to show that

$$f_{n,k} \;=\; f_{n-1,k}+f_{n-2,k-1} \quad \text{for} \quad n\ge 2, \tag{10.23}$$

provided we set $f_{n,k}=0$ when $k<0$. Let $F_n(y)=\sum_k f_{n,k}y^k$ and sum y^k times (10.23) over all k to obtain

$$F_n(y) \;=\; F_{n-1}(y)+yF_{n-2}(y) \quad \text{for } n\ge 2. \tag{10.24}$$

For $n=0$ we have only the empty sequence and for $n=1$ we have the two sequences 0 and 1. Thus, the initial conditions for (10.24) are $F_0(y)=1$ and $F_1(y)=1+y$. Multiplying (10.24) by x^n and summing over $n\ge 2$, we obtain

$$F(x,y)-F_0(y)-xF_1(y) \;=\; x\big(F(x,y)-F_0(y)\big)+x^2yF(x,y).$$

Thus

$$F(x,y) \;=\; \frac{1+xy}{1-x-x^2y}. \tag{10.25}$$

We are now ready to use (10.21). From (10.25),

$$F_y(x,y) \;=\; \frac{x(1-x-x^2y)-(1+xy)(-x^2)}{(1-x-x^2y)^2} \;=\; \frac{x}{(1-x-x^2y)^2}$$

and so

$$F_y(x,1) \;=\; \frac{x}{(1-x-x^2)^2}.$$

Thus

$$[x^n]\,F_y(x,1) \;=\; [x^{n-1}]\,\frac{1}{(1-x-x^2)^2}.$$

This can be expanded by partial fractions in various ways. The easiest method is probably to use the ideas and formulas in Appendix D (p. 387), which we now do. With $a,b=(1\pm\sqrt{5})/2$, as in Example 10.2, we have

$$\frac{1}{(1-x-x^2)^2} \;=\; \frac{1}{(1-ax)^2(1-bx)^2}.$$

We make use of the relations

$$a+b=1 \quad ab=-1 \quad \text{and} \quad a-b=\sqrt{5}.$$

Here are the calculations

$$\begin{aligned}\frac{1}{(1-ax)^2(1-bx)^2} &= \left(\frac{a/\sqrt{5}}{1-ax} - \frac{b/\sqrt{5}}{1-bx}\right)^2 \\ &= \frac{a^2/5}{(1-ax)^2} - \frac{2ab/5}{(1-ax)(1-bx)} + \frac{b^2/5}{(1-bx)^2} \\ &= \frac{a^2/5}{(1-ax)^2} + \frac{2a/5\sqrt{5}}{1-ax} - \frac{2b/5\sqrt{5}}{1-bx} + \frac{b^2/5}{(1-bx)^2}.\end{aligned}$$

Thus

$$\begin{aligned}[x^n]\, F_y(x,1) &= \frac{a^2}{5}\binom{-2}{n-1}(-a)^{n-1} + \frac{2a}{5\sqrt{5}}a^{n-1} - \frac{2b}{5\sqrt{5}}b^{n-1} + \frac{b^2}{5}\binom{-2}{n-1}(-b)^{n-1} \\ &= \frac{na^{n+1}}{5} + \frac{2a^n}{5\sqrt{5}} - \frac{2b^n}{5\sqrt{5}} + \frac{nb^{n+1}}{5}.\end{aligned}$$

Since $|b| < .62$, the last two terms in this expression are fairly small. In fact, we will show that w_n is the integer closest to

$$\frac{a^n}{5}\big(an + 2/\sqrt{5}\big).$$

Using the expression (10.15) for $\sum_k f_{n,k}$, the average number of ones is very close to

$$\frac{n}{a\sqrt{5}} + \frac{2}{5a^2}.$$

We must prove our claim about the smallness of the terms involving b. It suffices to show that their sum is less than $1/2$. Since $|b| = (\sqrt{5}-1)/2 < 1$, we have

$$\frac{2|b|^n}{5\sqrt{5}} \;\le\; \frac{2b}{5\sqrt{5}} \;<\; 0.12.$$

The term $n|b|^{n+1}/5$ is a bit more complicated. We study it as a function of n to find its maximum. Its derivative with respect to n is

$$\frac{|b|^{n+1}}{5} + \frac{n\ln|b|\;|b|^{n+1}}{5} \;=\; \frac{|b|^{n+1}}{5}(1 + n\ln|b|).$$

Since $-0.25 < \ln|b| < -0.2$, this is positive for $n \le 4$ and negative for $n \ge 5$. It follows that the term achieves its maximum at $n = 4$ or at $n = 5$. The values of these two terms are

$$4|b|^5/5 \;<\; |b|^5 \;<\; 0.1 \;\text{ and }\; 5|b|^6/5 \;<\; |b|^5 \;<\; 0.1,$$

proving our claim. □

Example 10.10 Leaves in trees What can we say about the number of leaves in n-vertex unlabeled RP-trees? We'll study the average number of leaves and the variance using (10.21) and (10.22).

Let $t_{n,k}$ be the number of unlabeled RP-trees having n vertices and k leaves and let $T(x,y)$ be $\sum_{n,k} t_{n,k}x^n y^k$. Using tools at our disposal, it is not easy to work out the generating function for $T(x,y)$. On the other hand, after you have read the next section, you should be able to show that

$$T(x,y) \;=\; xy + xT(x,y) + x(T(x,y))^2 + \cdots + x(T(x,y))^i + \cdots,$$

where $x(T(x,y))^i$ comes from building trees whose roots have degree i. We'll assume this has been done. Summing the geometric series in (10.25), we have

$$T(x,y) \;=\; xy + \frac{xT(x,y)}{1-T(x,y)}.$$

Clearing of fractions and rearranging:

$$(T(x,y))^2 - (1-x+xy)T(x,y) + xy \;=\; 0,$$

a quadratic equation in $T(,y)$ whose solution is

$$T(x,y) \;=\; \frac{1-x+xy \pm \sqrt{(1-x+xy)^2-4xy}}{2} \;=\; \frac{1-x+xy \pm \sqrt{(1+x-xy)^2-4x}}{2}.$$

Do we use the plus sign or the minus sign? Since there are no trees with no vertices $t_{0,0} = 0$. On the other hand,

$$t_{0,0} \;=\; T(0,0) \;=\; \frac{1 \pm \sqrt{1}}{2}$$

and so we want the minus sign. We finally have $T(x,y)$. Let's multiply be 2 to get rid of the annoying fraction:

$$2T(x,y) \;=\; 1-x+xy-\big((1+x-xy)^2-4x\big)^{1/2}.$$

Differentiating with respect to y, we have

$$2T_y(x,y) \;=\; x + x(1+x-xy)\big((1+x-xy)^2-4x\big)^{-1/2}$$

and

$$2T_{yy}(x,y) \;=\; -x^2\big((1+x-xy)^2-4x\big)^{-1/2} + x^2(1+x-xy)^2\big((1+x-xy)^2-4x\big)^{-3/2}.$$

Thus

$$\begin{aligned} 2T(x,1) &= 1-(1-4x)^{1/2}, \\ 2T_y(x,1) &= x + x(1-4x)^{-1/2}, \\ 2T_{yy}(x,1) &= -x^2(1-4x)^{-1/2} + x^2(1-4x)^{-3/2}. \end{aligned}$$

For $n > 2$ we have

$$\begin{aligned} 2[x^n]\,T(x,1) &= -(-4)^n\binom{1/2}{n}, \\ 2[x^n]\,T_y(x,1) &= [x^{n-1}]\,(1-4x)^{-1/2} \;=\; (-4)^{n-1}\binom{-1/2}{n-1}, \\ 2[x^n]\,T_{yy}(x,1) &= -[x^{n-2}]\,(1-4x)^{-1/2} + [x^{n-2}]\,(1-4x)^{-3/2} \\ &= -(-4)^{n-2}\binom{-1/2}{n-2} + (-4)^{n-2}\binom{-3/2}{n-2}. \end{aligned}$$

Let X_n be the number of leaves in a random n-vertex tree and suppose $n > 2$. Then

$$\begin{aligned} \mathbf{E}(X_n) \;&=\; \frac{2[x^n]\,T_y(x,1)}{2[x^n]\,T(x,1)} \;=\; \frac{\binom{-1/2}{n-1}}{-(-4)\binom{1/2}{n}} \\ &=\; \frac{\dfrac{(-1/2)(-3/2)\cdots(-1/2-(n-2))}{(n-1)!}}{4\dfrac{(1/2)(-1/2)\cdots(1/2-(n-1))}{n!}} \;=\; \frac{n!}{4(1/2)\,(n-1)!} \;=\; \frac{n}{2} \end{aligned}$$

and, recalling (10.22),

$$\begin{aligned}
\mathbf{E}(X_n^2) &= \frac{2[x^n]\,T_{yy}(x,1)}{2[x^n]\,T(x,1)} + \frac{2[x^n]\,T_y(x,1)}{2[x^n]\,T(x,1)} = \left(\frac{\binom{-1/2}{n-2}}{4^2\binom{1/2}{n}} - \frac{\binom{-3/2}{n-2}}{4^2\binom{1/2}{n}}\right) + \frac{n}{2} \\
&= \frac{\dfrac{(-1/2)\cdots(-1/2-(n-3))}{(n-2)!}}{4^2\dfrac{(1/2)\cdots(1/2-(n-1))}{n!}} - \frac{\dfrac{(-3/2)\cdots(-3/2-(n-3))}{(n-2)!}}{4^2\dfrac{(1/2)\cdots(1/2-(n-1))}{n!}} + \frac{n}{2} \\
&= \frac{n!}{4^2(1/2)(1/2-(n-1))\,(n-2)!} - \frac{n!}{4^2(1/2)(-1/2)\,n!} + \frac{n}{2} \\
&= \frac{n(n-1)}{4(3-2n)} + \frac{n(n-1)}{4} + \frac{n}{2} \\
&= \frac{n^2+n}{4} - \frac{n(n-1)}{4(2n-3)}.
\end{aligned}$$

Thus

$$\begin{aligned}
\mathbf{var}(X_n) &= \mathbf{E}(X_n^2) - (\mathbf{E}(X_n))^2 = \left(\frac{n^2+n}{4} - \frac{n(n-1)}{4(2n-3)}\right) - \frac{n^2}{4} \\
&= \frac{n}{4} - \frac{n(n-1)}{4(2n-3)} = \frac{n\big((2n-3)-(n-1)\big)}{4(2n-3)} = \frac{n(n-2)}{4(2n-3)}.
\end{aligned}$$

For large n this is nearly $n/8$.

We've shown that the average number of leaves in an RP-tree is $n/2$ and the variance in the number of leaves is about $n/8$. By Chebyshev's inequality (C.3) (p. 385), it follows that, in most large RP-trees, about half the vertices are leaves. More precisely:

$$\text{It is unlikely that } \frac{|(\text{number of leaves}) - n/2|}{\sqrt{n/8}} \text{ will be large.}$$

By Exercise 5.4.8 (p. 140), every N-vertex full binary tree has exactly $\frac{N+1}{2}$ leaves, very slightly larger than the average over all trees. Since a tree that has many edges out of nonleaf vertices will have more leaves, it would seem that a full binary tree should have relatively few leaves. What is going on? Random RP-trees must have many nonleaf vertices with only one child, counterbalancing those with many children so that the average comes out to be nearly two. □

***Example 10.11 Average distance to a leaf** What is the average distance to the leaf in a random full binary RP-tree?

Before answering this question, we need to say precisely what it means. If T is an unlabeled full binary RP-tree, let $d(T)$ be the sum of the distances from the root to each of the leaves of the tree. (The distance from the root to a leaf is the number of edges on the unique path joining them.) We want the average value of $d(T)/n$ over all unlabeled n leaf full binary RP-trees. This average can be important because many algorithms involve traversing such trees from the root to a leaf and the time required is proportional to the distance.

Let $D(x) = \sum d(T)x^{w(T)}$, where the sum ranges over all unlabeled full binary RP-trees T and $w(T)$ is the number of leaves in T. Let $B(x) = \sum x^{w(T)}$. By Example 10.5

$$B(x) = \frac{1-\sqrt{1-4x}}{2} \text{ and } b_n = -\frac{1}{2}\binom{\frac{1}{2}}{n}(-4)^n = \frac{1}{n}\binom{2n-2}{n-1}.$$

Suppose that T has more than one leaf. Let T_1 and T_2 be the two principal subtrees of T; that is, the two trees whose roots are the sons of the root of T. You should be able to show that

$$d(T) = w(T) + d(T_1) + d(T_2).$$

Multiply this by $x^{w(T)}$ and sum over all T with more than one leaf. Since $d(\bullet) = 0$ and $w(T) = w(T_1) + w(T_2)$, we have

$$\begin{aligned} D(x) &= \sum_{n>1} nb_n x^n + \sum_{T_1,T_2} d(T_1)x^{w(T_1)+w(T_2)} + \sum_{T_1,T_2} d(T_2)x^{w(T_1)+w(T_2)} \\ &= xB'(x) - x + D(x)B(x) + B(x)D(x). \end{aligned}$$

Thus

$$D(x) = \frac{xB'(x) - x}{1 - 2B(x)} = \frac{1}{\sqrt{1-4x}}\left(\frac{x}{\sqrt{1-4x}} - x\right) = \frac{x}{1-4x} - \frac{x}{\sqrt{1-4x}}.$$

It follows that

$$d_n = 4^{n-1} - \binom{-\frac{1}{2}}{n-1}(-4)^{n-1} = 4^{n-1} + \frac{n}{2}\binom{\frac{1}{2}}{n-1}(-4)^n = 4^{n-1} - nb_n$$

and so the average distance to a leaf is

$$\frac{4^{n-1}}{nb_n} - 1 = \frac{4^{n-1}}{\binom{2n-2}{n-1}} - 1.$$

Using Stirling's formula, it can be shown that this is asymptotic to $\sqrt{\pi n}$.

This number is fairly small compared to n. We could do much better by limiting ourselves to averaging over certain subclasses of binary RP-trees. For example, we saw in Chapter 8 that if the distances to the leaves of the tree are all about equal, then the average and *largest* distances are both only about $\log_2 n$. Thus, when designing algorithms that use trees as data structures, restricting the shape of the tree could lead to significant savings. Good information storage and retrieval algorithms are designed on this basis. □

*Example 10.12 The average time for Quicksort We want to find out how long it takes to sort a list using Quicksort. Quicksort was discussed briefly in Chapter 8. We'll review it here. Given a list $a_1, a_2, \ldots, a_n$, Quicksort selects an element x, divides the list into two parts (greater and less than x) and sorts each part by calling itself. There are two problems. First, we haven't been specific enough in our description. Second, the time Quicksort takes depends on the order of the list and the way x is chosen at each call. To avoid the dependence on order, we will average over all possible arrangements. We now give a more specific description using $x = a_1$. Given a list $a_1, a_2, \ldots, a_n$ of distinct elements, we create a new list $s_1, s_2, \ldots, s_n$ with the following properties.

(a) For some $1 \le k \le n$, $s_k = a_1$.

(b) $s_i < a_1$ for $i < k$ and $s_i > a_1$ for $i > k$.

(c) The relative order of the elements in the two sublists is the same as in the original list; i.e., if $s_i = a_p$, $s_j = a_q$ and either $i < j < k$ or $k < i < j$, then $p < q$.

It turns out that this can be done with $n - 1$ comparisons. We now apply Quicksort recursively to $s_1, \ldots, s_{k-1}$ and to $s_{k+1}, \ldots, s_n$.

Let q_n be the average number of comparisons needed to Quicksort an n long list. Thus $q_1 = 0$. We define $q_0 = 0$ for convenience later.

Note that k is the position of a_1 in the sorted list. Since the original list is random, all values of k from 1 to n are equally likely. By analyzing the algorithm carefully, it can be shown that all orderings of $s_1, \ldots, s_{k-1}$ are equally likely as are all orderings of $s_{k+1}, \ldots, s_n$. (We will not do this.) Thus, given k, it follows that the average length of time needed to sort both $s_1, \ldots, s_{k-1}$ and $s_{k+1}, \ldots, s_n$ is $q_{k-1} + q_{n-k}$.

Averaging over all possible values of k and remembering to include the original $n - 1$ comparisons, we obtain

$$q_n = n - 1 + \frac{1}{n}\sum_{k=1}^{n}\big(q_{k-1} + q_{n-k}\big) = n - 1 + \frac{2}{n}\sum_{j=0}^{n-1} q_j,$$

which is valid for $n > 0$.

To solve this recursion by generating functions, we should let $Q(x) = \sum q_n x^n$ and use the recursion to get a relation for $Q(x)$. If we simply substitute, we obtain

$$Q(x) \;=\; q_0 + \sum_{n=1}^{\infty} \left(n - 1 + \frac{2}{n} \sum_{j=0}^{n-1} q_j \right) x^n. \tag{10.26}$$

If we try to manipulate this to simplify the double sum over n and j of $2q_j x^n/n$, we will run into problems because of the n in the denominator. How can we deal with this?

One approach would be to multiply the original recursion by n before we use it. Another approach, which it turns out is equivalent, is to differentiate (10.26) with respect to x. Which is better? The latter is easier when we have a denominator as simple as n, but the former may be better when we have more complicated expressions. We use the latter approach. Differentiating (10.26), we have

$$\begin{aligned} Q'(x) \;&=\; \sum_{n=1}^{\infty} \left((n-1)n + 2 \sum_{j=0}^{n-1} q_j \right) x^{n-1} \;=\; \sum_{n=1}^{\infty} n(n-1) x^{n-1} + 2 \sum_{n=1}^{\infty} \sum_{j=0}^{n-1} q_j x^{n-1} \\ &=\; x \left(\frac{1}{1-x} \right)'' + 2 \sum_{k=0}^{\infty} \sum_{j=0}^{k} q_j x^k \;=\; \frac{2x}{(1-x)^3} + 2Q(x) \, \frac{1}{1-x}, \end{aligned}$$

where $Q(x)/(1-x)$ follows either by recognizing that we have a convolution or by applying Exercise 10.1.6 (p. 274).

Rearranging, we see that we must solve the differential equation

$$Q'(x) - 2(1-x)^{-1} Q(x) \;=\; 2x(1-x)^{-3}, \tag{10.27}$$

which is known as a linear first order differential equation. This can be solved by standard methods from the theory of differential equations. We leave it as an exercise to show that the solution is

$$Q(x) \;=\; \frac{-2\ln(1-x) - 2x + C}{(1-x)^2}, \tag{10.28}$$

where the constant C must be determined by an initial condition. Since $Q(0) = q_0 = 0$, we have $C = 0$.

Using the Taylor series

$$-\ln(1-x) \;=\; \sum \frac{x^k}{k}$$

and some algebra, one eventually obtains

$$q_n = 2(n+1) \sum_{k=1}^{n} \frac{1}{k} - 4n. \tag{10.29}$$

Again, details are left as an exercise.

Using Riemann sum approximations, we have

$$\sum_{k=2}^{n} \frac{1}{k} \;<\; \int_1^n \frac{dx}{x} \;<\; \sum_{k=1}^{n-1} \frac{1}{k},$$

from which it follows that the summation in (10.29) equals $\ln n + O(1)$. It follows that

$$q_n \;=\; 2n \ln n + O(n) \;\text{ as } n \to \infty. \tag{10.30}$$

This is not quite as small as the result $n \log_2 n$ that we obtained for worst case merge sorting of a list of length $n = 2^k$; however, merge sorting requires an extra array but Quicksort does not because the array $s_1, \ldots, s_n$ can simply replace the array $a_1, \ldots, a_n$. (Actually, merge sorting can be done "in place" if more time is spent on merging. The Batcher sort is an in place merge sort.) You might like to compare this with Exercise 8.2.10 (p. 238), where we obtained an estimate of $1.78\, n \ln n$ for q_n. □

Exercises

10.3.1. Let $D(x)$ be the "exponential" generating function for the number of derangements as in Example 10.7. You'll use (10.20) to derive a linear differential equation with polynomial coefficients for $D(x)$. Then you'll equate coefficients to get a recursion for D_n.

(a) Differentiate $(1-x)D(x) = e^{-x}$ and the use $e^{-x} = (1-x)D(x)$ to eliminate e^{-x}.

(b) Equate coefficients to obtain $D_{n+1} = n(D_n + D_{n-1})$ for $n > 0$. What are the initial conditions?

10.3.2. A "path" of length n is a sequence $0 = u_0, u_1, \ldots, u_n = 0$ of nonnegative integers such that $u_{k+1} - u_k \in \{-1, 0, 1\}$ for $k < n$. Let a_n be the number of such paths of length n The OGF for a_n can be shown to be $A(x) = (1 - 2x - 3x^2)^{-1/2}$.

(a) Show that $(1 - 2x - 3x^2)A'(x) = (1 + 3x)A(x)$.

(b) Obtain the recursion

$$(n+1)a_{n+1} = (2n+1)a_n + 3na_{n-1} \quad \text{for} \quad n > 0.$$

What are the initial conditions?

(c) Use the general binomial theorem to expand $(1-(2x+3x^2))^{-1/2}$ and then the binomial theorem to expand $(2x + 3x^2)^k$. Finally look at the coefficient of x^n to obtain a_n as a sum involving binomial coefficients.

10.3.3. Fill in the steps in the derivation of the average time formula for Quicksort:

(a) Solve (10.27) to obtain (10.28) by using an integrating factor or any other method you wish.

(b) Obtain (10.29) from (10.28).

10.3.4. In Exercise 10.2.6, you derived the formula

$$T(x) = \frac{1 - x - \sqrt{1 - 2x - 3x^2}}{2x}.$$

Use the methods of this section to derive a recursion for t_n that is simpler than the summation in Exercise 10.2.6(a).
Hint. Since the manipulations involve a fair bit of algebra, it's a good idea to check your recursion for t_n by comparing it with actual value for small n. They can be determined by constructing the trees.

10.4 The Rules of Sum and Product

Before the 1960's, combinatorial constructions and generating function equations were, at best, poorly integrated. A common route to a generating function was:

1. Obtain a combinatorial description of how to construct the structures of interest; e.g., the recursive description of unlabeled full binary RP-trees.
2. Translate the combinatorial description into equations relating elements of the sequence that enumerate the objects; e.g., $b_n = \sum_{k=1}^{n-1} b_k b_{n-k}$, for $n > 1$ and $b_1 = 1$.
3. Introduce a generating function for the sequence and substitute the equations into the generating function. Apply algebraic manipulation.
4. The result is a relation for the generating function.

From the 1960's on, various people have developed methods for going directly from a combinatorial construction to a generating function expression, eliminating Steps 2 and 3. These methods often

allow us to proceed from Step 1 directly to Step 4. The Rules of Sum and Product for generating functions are basic tools in this approach. We study them in this section.

So far we have been thinking of generating functions as being associated with a sequence of numbers $a_0, a_1, \ldots$ which usually happen to be counting something. It is often helpful to think more directly about what is being counted. For example, let $\mathcal{B}$ be the set of unlabeled full binary RP-trees. For $B \in \mathcal{B}$, let $w(B)$ be the number of leaves of B. Then b_n is simply the number of $B \in \mathcal{B}$ with $w(B) = n$ and so

$$\sum_{B \in \mathcal{B}} x^{w(B)} \;=\; \sum_n b_n x^n = B(x). \tag{10.31}$$

We say that $B(x)$ counts unlabeled full binary RP-trees by number of leaves. It is sometimes convenient to refer to the generating function by the set that is associated with it. In this case, the set is $\mathcal{B}$ so we use the notation $\mathsf{G}_{\mathcal{B}}(x)$ or simply $\mathsf{G}_{\mathcal{B}}$. Thus, instead of asking for the generating function for the b_n's, we can just as well ask for the generating function for unlabeled full binary RP-trees (by number of leaves). Similarly, instead of asking for the generating function for F_n, we can ask for the generating function for sequences of zeroes and ones with no adjacent ones (by the length of the sequence). When it is clear, we may omit the phrase "by number of leaves," or whatever it is we are counting things by. We could also keep track of more than one thing simultaneously, like the length of a sequence *and* the number of ones. We won't pursue that now.

As noted above, if $\mathcal{T}$ is some set of structures (e.g., $\mathcal{T} = \mathcal{B}$), we let $\mathsf{G}_{\mathcal{T}}$ be the generating function for $\mathcal{T}$, with respect to whatever we are counting the structures in $\mathcal{T}$ by (e.g., leaves in (10.31)).

The Rule of Sum for generating functions is nothing more than a restatement of the Rule of Sum for counting that we developed in Chapter 1. The Rule of Product is a bit more complex. At this point, you may find it helpful to look back at the Rules of Sum and Product for counting: Theorem 1.2 (p. 6) and Theorem 1.3 (p. 8).

Theorem 10.3 Rule of Sum *Suppose a set $\mathcal{T}$ of structures can be partitioned into sets $\mathcal{T}_1, \ldots, \mathcal{T}_j$ so that each structure in $\mathcal{T}$ appears in exactly one $\mathcal{T}_i$. It then follows that*

$$\mathsf{G}_{\mathcal{T}}(x) \;=\; \mathsf{G}_{\mathcal{T}_1}(x) + \cdots + \mathsf{G}_{\mathcal{T}_j}(x).$$

The Rule of Sum remains valid when the number of blocks in the partition $\mathcal{T}_1, \mathcal{T}_2, \ldots$ is infinite.

Theorem 10.4 Rule of Product *Let w be a function that counts something in structures. Suppose each T in a set $\mathcal{T}$ of structures is constructed from a sequence $T_1, \ldots, T_k$ of k structures such that*

(*i*) *the possible structures T_i for the ith choice may depend on previous choices, but the generating function for them does not depend on previous choices,*

(*ii*) *each structure arises in exactly one way in this process and*

(*iii*) *if the structure T comes from the sequence $T_1, \ldots, T_k$, then*

$$w(T) \;=\; w(T_1) + \ldots + w(T_k).$$

It then follows that

$$\mathsf{G}_{\mathcal{T}}(x) \;=\; \sum_{T \in \mathcal{T}} x^{w(T)} \;=\; \mathsf{G}_1(x) \cdots \mathsf{G}_k(x), \tag{10.32}$$

where G_i is the generating function for the possible choices for the ith structure.

The Rule of Product remains valid when the number of steps is infinite.

As with the Rule of Product for counting, the available choices for the ith step may depend on the previous choices, but the generating function must not. If the choices at the ith step do not depend on the previous choices, we can think of $\mathcal{T}$ as simply a Cartesian product $\mathcal{T}_1 \times \cdots \times \mathcal{T}_k$.

The additivity condition (iii) is needed to insure that multiplication works correctly, namely

$$x^{w(T)} \;=\; x^{w(T_1)} \cdots x^{w(T_k)}.$$

Weights that count things (e.g., leaves in trees, cycles in a permutation, size of set being partitioned) usually satisfy (iii). This is not always the case; for example, counting the number of distinct things (e.g., cycle lengths in a permutation) is usually not additive. Weights dealing with a maximum (e.g., longest path from root to leaf in a tree, longest cycle in a permutation) do *not* satisfy (iii).

Proof: We will prove (10.32) by induction on k, starting with $k = 2$. The induction step is practically trivial—simply group the first $k-1$ choices together as one choice, apply the theorem for $k = 2$ to this grouped choice and the kth choice, and then apply the theorem for $k-1$ to the grouped choice.

The proof for $k = 2$ can be obtained by applying of the Rules of Sum and Product for counting as follows. Let $t_{i,j}$ be the number of ways to choose the ith structure so that it contains exactly j of the objects we are counting; that is, the number of ways to choose T_i so that $w(T_i) = j$. The number of ways to choose T_1 so that it contains j objects AND then choose T_2 so that together T_1 and T_2 contain n objects is $t_{1,j}\, t_{2,n-j}$. Thus, the total number of structures in $\mathcal{T}$ that contain exactly n objects is

$$\sum_{j=0}^{n} t_{1,j}\, t_{2,n-j}.$$

Multiplying by x^n, summing over n and recognizing that we have a convolution, we obtain (10.32) for $k = 2$.

Compare the proof we have just given for $k = 2$ with the following. By hypotheses (ii) and (iii) of the theorem,

$$\sum_{T \in \mathcal{T}} x^{w(T)} \;=\; \sum_{T_1 \in \mathcal{T}_1} x^{w(T_1)} \Bigg(\sum_{T_2 \in \mathcal{T}_2} x^{w(T_2)} \Bigg).$$

By hypothesis (i), the inner sum equals G_2 even though $\mathcal{T}_2$ may depend on T_1. Thus the above expression becomes $\mathsf{G}_1\mathsf{G}_2$. While this might seem almost magical, it's a perfectly valid proof. The lesson here is that it's often easier to sum over structures than to sum over indices.

Passing to the infinite case in the theorems is essentially a matter of taking a limit. We omit the proof. □

Example 10.13 Binomial coefficients Let's apply these theorems to enumerating binomial coefficients. Our structures will be subsets of $\underline{n}$ and we will be keeping track of the number of elements in a subset; i.e., $w(S) = |S|$, the number of elements in S. We form all subsets exactly once by a sequence of n choices. The ith choice will be either $\emptyset$ (the empty set) or the set $\{i\}$. The union of our choices will be a subset. The Rule of Product can be applied. Since $w(\emptyset) = 0$ and $w(\{i\}) = 1$, $\mathsf{G}_i(x) = 1 + x$ by the Rule of Sum. Thus the generating function for subsets of $\underline{n}$ by cardinality is $(1+x)\cdots(1+x) = (1+x)^n$. Compare this with the derivation in Example 1.14 (p. 19). Because this problem is so simple and because you are not familiar with using our two theorems, you may find the derivation in Example 1.14 easier than the one here. Read on. □

Example 10.14 Counting unlabeled RP-trees Let's look at unlabeled RP-trees from this new vantage point. If a tree has more that one vertex, let $s_1, \ldots, s_k$ be the sons of the root from left to right. We can describe such a tree by listing the k subtrees $T_1, \ldots, T_k$ whose roots are $s_1, \ldots, s_k$. This gives us a k-tuple. Note that T has as many leaves as $T_1, \ldots, T_k$ together. In fact, if you look back to the start of Chapter 9, you will see that this is nothing more nor less than the definition we gave there.

Let $B(x)$ be the generating function for unlabeled full binary unlabeled RP-trees by number of leaves. By the previous paragraph, an unlabeled full binary RP-tree is either one vertex OR a 2-tuple of unlabeled full binary RP-trees (joined to a new root). Applying the Rules of Sum and Product with $j = k = 2$, we have

$$\mathsf{G}_{\mathcal{B}}(x) \;=\; x + \mathsf{G}_{\mathcal{B}}(x)\,\mathsf{G}_{\mathcal{B}}(x),$$

which can also be written

$$B(x) \;=\; x + B(x)B(x).$$

This is much easier than deriving the recursion first—compare this derivation with the one in Example 10.5 (p. 279).

Now let's count arbitrary unlabeled RP-trees. In this case, we cannot count them by leaves because there are an infinite number of trees with just one leaf: any path is such a tree. We'll count them by vertices. Let $T(x)$ be the generating function. Proceeding as in the previous paragraph, we say that such a tree is either a single vertex, OR one tree, OR a 2-tuple of trees, OR a 3-tuple of trees, and so on. Thus we (incorrectly) write $T(x) = x + T(x) + T^2(x) + \cdots$. Why is this wrong? We did not apply the Rule of Product correctly. The number of vertices in a tree T is *not* equal to the total number of vertices in the k-tuple $(T_1, \ldots, T_k)$ that comes from the sons of the root: We forgot that there is one more vertex, the root of T.

Let's do this correctly. Instead of a k-tuple of trees, we have a vertex AND a k-tuple of trees. Thus a tree is either a single vertex, OR a single vertex AND a tree, OR a single vertex AND a 2-tuple of trees, and so on. Now we get (correctly)

$$T(x) \;=\; x + xT(x) + xT^2(x) + \cdots \;=\; \frac{x}{1 - T(x)},$$

by the Rules of Sum and Product and the formula for a sum of a geometric series. Multiplying by $1 - T(x)$, we have $T(x) - T^2(x) = x$, which is the same as the equation for $B(x)$. Thus

Theorem 10.5 *The number of n vertex unlabeled RP-trees equals the number of n leaf unlabeled full binary RP-trees.*

This was proved in Example 7.9 (p. 206) by showing that the numbers satisfied the same recursion and in Exercise 9.3.12 (p. 266) by giving a bijection.

You should be able to derive $T(x) = x + T(x)^2$ directly from the second definition of RP-trees in Example 7.9 (p. 206) and hence prove the theorem this way.

We've looked at two extremes: full binary trees (all nonleaf vertices have exactly 2 children) and arbitrary trees (nonleaf vertices can have any number of children). We can study trees in between these two extremes. Let D be a set of positive integers. Let $\mathcal{D}$ be those unlabeled RP-trees where the number of children of each vertex lies in D. The two extremes correspond to $D = \{2\}$ and $D = \{1, 2, 3, \ldots\}$. If we count these trees by number of vertices, you should be able to show that

$$\mathsf{G}_{\mathcal{D}}(x) \;=\; x + \sum_{d \in \mathcal{D}} x\mathsf{G}_{\mathcal{D}}(x)^d.$$

In general, we cannot solve this equation; however, we can simplify the sum if the elements of $\mathcal{D}$ lie in an arithmetic progression. Our two extremes are examples of this. For another example, suppose $\mathcal{D}$ is the set of positive odd integers. Then the sum is a geometric series with first term $x\mathsf{G}_{\mathcal{D}}(x)$ and ratio $\mathsf{G}_{\mathcal{D}}(x)^2$. After some algebra, one obtains a cubic equation for $\mathsf{G}_{\mathcal{D}}(x)$. We won't pursue this. □

Example 10.15 Balls in boxes Problems that involve placing unlabeled balls into labeled boxes (or, equivalently, problems that involve compositions of integers), are often easy to do using the Rules of Sum and Product. Let $\mathcal{T}_i$ be the set of possible ways to put things into the ith box. Let $\mathsf{G}_{\mathcal{T}_i}$ be the generating function which is keeping track of the things in the ith box. Suppose that what can be placed into one box is not dependent on what is placed in other boxes. The Rule of Product (in the Cartesian product form), tells us that we can simply multiply the $\mathsf{G}_{\mathcal{T}_i}$'s together.

How many ways can we put unlabeled balls into k labeled boxes so that no box is empty? Since there is exactly one way to place j balls in a box for every $j > 0$ and no ways if $j = 0$ (since the box may not be empty), we have

$$\mathsf{G}_{\mathcal{T}_i}(x) \;=\; 1x^0 + 1x^1 + 1x^2 + \cdots \;=\; \sum_{j=1}^{\infty} x^j = \frac{x}{1-x}$$

for all i. By the Rule of Product, the generating function is

$$\frac{x}{1-x} \cdots \frac{x}{1-x} \;=\; x^k(1-x)^{-k}.$$

Since

$$x^k(1-x)^{-k} \;=\; x^k \sum \binom{-k}{i}(-x)^i \;=\; \sum \binom{k+i-1}{i} x^{k+i},$$

it follows that the number of ways to distribute n unlabeled balls is $\binom{n-1}{n-k} = \binom{n-1}{k-1}$, which you found in Exercise 1.5.4 (p. 38).

How many solutions are there to the equation $z_1 + z_2 + z_3 = n$ where z_1 is an odd positive integer and z_2 and z_3 are nonnegative integers not exceeding 10? We can think of this as placing balls into boxes where z_i balls go into the ith box. Since

$$\mathsf{G}_{\mathcal{T}_1}(x) \;=\; x + x^3 + x^5 + \cdots \;=\; x\big(1 + x^2 + (x^2)^2 + (x^2)^3 + \cdots\big) \;=\; \frac{x}{1-x^2}$$

and

$$\mathsf{G}_{\mathcal{T}_2}(x) \;=\; \mathsf{G}_{\mathcal{T}_3}(x) \;=\; 1 + x + \cdots + x^{10} \;=\; \frac{1-x^{11}}{1-x},$$

it follows that the generating function is

$$\frac{x}{1-x^2}\,\frac{1-x^{11}}{1-x}\,\frac{1-x^{11}}{1-x}.$$

There isn't a nice formula for the coefficient of x^n.

What if we allow positive integer coefficients in our equation? For example, how many solutions are there to $z_1 + 2z_2 + 3z_3 = n$ in nonnegative integers? In this case, put z_1 balls in the first box, $2z_2$ balls in the second and $3z_3$ balls in the third. Since the number of balls in box i is a multiple of i, $\mathsf{G}_{\mathcal{T}_i}(x) = 1/(1-x^i)$. By the Rule of Product $\mathsf{G}_{\mathcal{T}}(x) = 1/((1-x)(1-x^2)(1-x^3))$. This result can be thought of as counting partitions of the number n where z_i is the number of parts of size i. By extending this idea, it follows that, if $p(n)$ is the number of partitions of the integer n, then

$$\sum_{n=0}^{\infty} p(n)x^n \;=\; \frac{1}{1-x}\,\frac{1}{1-x^2}\,\frac{1}{1-x^3} \cdots \;=\; \prod_{i=1}^{\infty}(1-x^i)^{-1}. \;\square$$

So far we have only used the Rules of Sum and Product for single variable generating functions. We need not limit ourselves in this manner. As we will explain:

Observation *The Rules of Sum and Product apply to generating functions with any number of variables.*

Suppose we are keeping track of m different kinds of things. Replace w by $\mathbf{w}$, an m long vector of integers. Then $\vec{x}^{\vec{w}} = x_1^{w_1} \cdots x_m^{w_m}$. For example, if we count words by the number of vowels, the number of consonants and the length of the word, $\mathbf{w}$ will be a 3 long vector—one component for number of vowels, one for number of consonants and one for total number of letters. In that case, the variables will also form a 3 long vector $\mathbf{x}$. We can replace (10.31) with

$$\sum_{B \in \mathcal{B}} \mathbf{x}^{\mathbf{w}(B)} \;=\; B(\mathbf{x}),$$

where, as we already said, $\mathbf{x}^{\mathbf{w}}$ means $x_1^{w_1} \cdots x_m^{w_m}$. The condition on w in the Rule of Product becomes

$$\mathbf{w}(T) \;=\; \mathbf{w}(T_1) + \ldots + \mathbf{w}(T_k).$$

Of course, we could choose other indices besides $1, \ldots, m$ for our vectors and even replace some of the x_i's with other letters. In the next example, we find it convenient to use $\mathbf{x} = (x_0, x_1)$.

Example 10.16 Strings of zeroes and ones Let's look at strings of zeroes and ones. It will be useful to have a shorthand notation for writing down strings. The empty string will be denoted by λ. If s is a string, then $(s)^k$ stands for the string $ss \ldots s$ that consists of k copies of s and $(s)^*$ stands for the set of strings that consist of any number of copies of s, i.e., $(s)^* = \{\lambda, s, (s)^2, (s)^3, \ldots\}$. When s is simply 0 or 1, we usually omit the parentheses. Thus we write 0^* and 1^k instead of $(0)^*$ and $(1)^k$.

The sequences counted by the Fibonacci numbers, namely those which contain no adjacent ones, can be described by

$$\mathcal{F} \;=\; 0^* \cup (0^*\ 1\ Z^*\ 0^*) \quad \text{where} \quad Z \;=\; 0^*\ 01.$$

This means

(a) any number of zeroes OR

(b) any number of zeroes AND a one AND any number of sequences of the form Z to be described shortly AND any number of zeroes.

A sequence of the form Z is any number of zeroes AND a zero AND a one. You should convince yourself that $\mathcal{F}$ does indeed give exactly those sequences which contain no adjacent ones. As you can guess from the ANDs and ORs above, this is just the right sort of situation for the Rules of Sum and Product.

What good does such a representation do us?

Observation *If this representation gives every pattern in exactly one way, we can mechanically use the Rules of Sum and Product to obtain a generating function.*

For a union (i.e., $\cup$ or $\{\cdots\}$), we are dealing with OR, so the Rule of Sum applies. When symbols appear to be multiplied it means first one thing AND then another, so we can apply the Rule of Product. For a set $\mathcal{S}$, the notation $\mathcal{S}^*$ means any number of copies of things in $\mathcal{S}$. For example, $\{0,1\}^*$ is the set of all strings of zeroes and ones, including the empty string. If there is a *unique* way to decompose elements of $\mathcal{S}^*$ into elements in $\mathcal{S}$, then

$$\mathsf{G}_{\mathcal{S}^*} \;=\; \mathsf{G}_{\emptyset} + \mathsf{G}_{\mathcal{S}} + \mathsf{G}_{\mathcal{S}\times\mathcal{S}} + \mathsf{G}_{\mathcal{S}\times\mathcal{S}\times\mathcal{S}} + \cdots \;=\; \sum_{k=0}^{\infty} (\mathsf{G}_{\mathcal{S}})^k \;=\; \frac{1}{1-\mathsf{G}_{\mathcal{S}}}.$$

What's "unique" decomposition mean? When $\mathcal{S} = \{0,1\}$, every string of zeroes and ones has a unique decomposition—just look at each element of the string one at a time. When $\mathcal{S} = \{0, 01, 11\}$ we still have unique decomposition; for example, 110001111101 decomposes uniquely as 11-0-01-11-11-01.

We leave it to you to verify that our representation for $\mathcal{F}$ gives all the patterns exactly once. Let x_0 keep track of zeroes and x_1 keep track of ones; that is, the coefficient of $x_0^n x_1^m$ in $\mathsf{G}_{\mathcal{F}}(x_0, x_1)$ will be the number sequences in $\mathcal{F}$ that have n zeroes and m ones. We have

$$\begin{aligned}
\mathsf{G}_{0^*} \;&=\; \frac{1}{1-\mathsf{G}_0} = \frac{1}{1-x_0} = (1-x_0)^{-1} \\
\mathsf{G}_{Z^*} \;&=\; \mathsf{G}_{(0^*\,01)^*} \;=\; \frac{1}{1-\mathsf{G}_{0^*\,01}} = \frac{1}{1-(1-x_0)^{-1}x_0x_1} \;=\; \frac{1-x_0}{1-x_0-x_0x_1} \\
\mathsf{G}_{\mathcal{F}} \;&=\; \frac{1}{1-x_0} + \frac{1}{1-x_0}\, x_1\, \frac{1-x_0}{1-x_0-x_0x_1}\, \frac{1}{1-x_0} \;=\; \frac{1+x_1}{1-x_0-x_0x_1}.
\end{aligned}$$

We can use this representation to describe and count other sequences; however, the problem can get tricky if we are counting sequences that must avoid patterns more complicated than 11. There are various ways to handle the problem. One method is by the use of sets of recursions, which we'll discuss in the next chapter. Sequences that can be described in this fashion (we haven't said precisely what that means) are called *regular sequences*. They are, in fact, the strings that can be produced by regular grammars, which we saw in Section 9.2 were the strings that can be recognized by finite automata. See Exercise 10.4.19 (p. 304) for a definition and the connection with automata. There is a method for translating finite automata into recursions. We'll explore this in Example 11.2 (p. 310). □

We close this section with an example which combines the Rules of Sum and Product with some techniques for manipulating generating functions.

*Example 10.17 Counting certain spanning trees Let G be a simple graph with $V = \underline{n} \cup \{0\}$ and the $2n-1$ edges $\{i, i+1\}$ $(1 \le i < n)$ and $\{0, j\}$ $(1 \le j \le n)$. (Draw a picture!) How many spanning trees does G have? We'll call the number r_n.

To begin with, what does a spanning tree look like? An arbitrary spanning tree can be built as follows. First, select some of the edges $\{i, i+1\}$ $(1 \le i < n)$. This gives a graph H with vertex set $\underline{n}$. (Some vertices may not be on any edges.) For each component C of H, select a vertex j in C and add the edge $\{0, j\}$ to our collection. Convince yourself that this procedure gives all the trees.

We can imagine this in a different way. Let $\mathcal{T}$ be the set of rooted trees of the following form. For each $k > 0$, let $V = \underline{k} \cup \{0\}$ and let 0 be the root. The tree contains the $k-1$ edges $\{i, i+1\}$ $(1 \le i < k)$ and one edge of the form $\{0, j\}$ for some $1 \le j \le k$. Join together an ordered list of trees in $\mathcal{T}$ by merging their roots into one vertex and relabeling their nonroot vertices $1, 2, \ldots$ in order as shown in Figure 10.1. This process produces each spanning tree exactly once.

What we have just described is the perfect setup for the Rules of Sum (on k) and Product (of $\mathcal{T}$ with itself k times) when we count the number of vertices other than the vertex 0. Thus, recalling the definition of r_n at the start of the example,

$$R \;=\; \sum_{k=1}^{\infty} (\mathsf{G}_{\mathcal{T}})^k \;=\; \sum_{k=1}^{\infty} T^k = \frac{T}{1-T}.$$

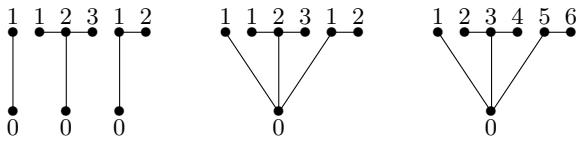


Figure 10.1 Building a spanning tree from pieces. The pieces on the left are assembled to give the middle figure and are then relabeled to give the right-hand figure.

How many trees in $\mathcal{T}$ have k nonroot vertices? Exactly k, one for each choice of a vertex to connect to the root. Thus $T(x) = \sum_{k=1}^{\infty} kx^k$. We can evaluate this sum by using derivatives as discussed in the previous section:

$$\begin{aligned} T(x) &= \sum_{k=1}^{\infty} kx^k = \sum_{k=0}^{\infty} kx^k = \sum_{k=0}^{\infty} x\frac{d(x^k)}{dx} \\ &= x\frac{d}{dx}\left(\sum_{k=0}^{\infty} x^k\right) = x\frac{d\big((1-x)^{-1}\big)}{dx} = \frac{x}{(1-x)^2}. \end{aligned}$$

Combining these results gives us

$$R(x) = \frac{\dfrac{x}{(1-x)^2}}{1 - \dfrac{x}{(1-x)^2}} = \frac{x}{1-3x+x^2}. \tag{10.33}$$

What can we do now to get values for r_n? We have two choices: (a) expand by partial fractions to get an exact value or (b) manipulate (10.33) using the ideas of the previous section to obtain a recursion. By partial fractions, r_n is the integer closest to $\alpha^n/\sqrt{5}$, where $\alpha = (3+\sqrt{5})/2$, which gives us a quick, accurate approximation to r_n for large n. We leave the calculations to you and turn our attention to deriving a recursion.

Clearing of fractions in (10.33) and equating the coefficients of x^n on both sides of the resulting equation gives the recursion

$$r_0 = 0, \quad r_1 = 1 \quad \text{and} \quad r_n = 3r_{n-1} - r_{n-2} \text{ for } n \geq 2, \tag{10.34}$$

which makes it fairly easy to build a table of r_n.

Can you prove (10.34) directly; i.e., without using generating functions? It's a bit tricky. With some thought and experimentation, you may be able to discover the argument. □

Exercises

10.4.1. Let $\mathcal{T}$ be a collection of structures. Suppose that $\mathbf{w}(T) \neq \mathbf{0}$ for all $T \in \mathcal{T}$. Prove the following results.

(a) The generating function for k-lists of structures, with repetitions allowed, is $(\mathsf{G}_{\mathcal{T}})^k$.

(b) The generating function for lists of structures, with repetitions allowed, is $(1-\mathsf{G}_{\mathcal{T}})^{-1}$. Here lists of any length are allowed, including the empty list.

(c) If T is a generating function, let $F^{[k]}$ denote the result of replacing all the variables by their kth powers. For example, $F^{[k]}(x,y) = F(x^k, y^k)$. Show that the generating function for sets of structures, where each structure must come from $\mathcal{T}$ is

$$\exp\left(\sum_{k=1}^{\infty} (-1)^{k-1} \big(\mathsf{G}_{\mathcal{T}}\big)^{[k]}/k\right).$$

Hint. Show that the answer is

$$\prod_{T\in\mathcal{T}}(1+\mathbf{x}^{\mathbf{w}(T)}),$$

replace $(1+\mathbf{x}^{\mathbf{w}(T)})$, with $\exp\bigl(\ln(1+\mathbf{x}^{\mathbf{w}(T)})\bigr)$, expand the logarithm by Taylor's Theorem and rearrange the terms.

(d) Show that generating function for multisets of structures is

$$\exp\biggl(\sum_{k=1}^{\infty}\bigl(\mathsf{G}_{\mathcal{T}}\bigr)^{[k]}/k\biggr).$$

10.4.2. Return to Exercise 10.2.4 (p. 280). There we counted the number of sequences of zeros, ones and twos with no adjacent ones and no adjacent twos. Show that part (a) of that exercise can be rewritten as follows.
A sequence of the type we want is either

- an alternating sequence of ones and twos OR
- a sequence of the type we want AND a zero AND an alternating sequence of ones and twos.

Here the alternating sequence may be empty. Use this characterization to deduce an equation that can be solved for $S(x)$

10.4.3. Using the notation introduced in Example 10.16, write out expressions for strings satisfying the following properties. Do this in such a way that each string is generated uniquely and then use your representation to get the generating function for the number of patterns of length n. Finally, obtain a recursion from your generating function. Remember to include initial conditions for the recursion.

(a) Strings of zeroes, ones and twos that have do not have the pattern 02 somewhere.
Hint. Except possibly for a run of twos at the very start of the string, every 2 must be preceded by a 1 or a 2.

(b) Strings of zeroes and ones such that each string of ones is followed by a string of at least k zeroes; i.e., if it starts with a string of zeroes, that can be of any length, but every other string of zeroes must have length at least k. Use the notation 0^k to stand for a string of k zeroes.

(c) Strings of zeroes and ones such that each maximal string of ones (i.e., its ends are the ends of the sequence and/or zeroes) has odd length.

10.4.4. Let q_n be the number of partitions of n with no repeated parts allowed and, as usual, let p_n be all partitions of n. Let $q_0 = 1$.

(a) Show that

$$Q(x)=\sum_{n=0}^{\infty}q_n x^n=\prod_{i=1}^{\infty}(1+x^i).$$

(b) Let $P(x)$ be the generating function for partitions of a number. Show that $Q(-x)\,P(x)=1$. Equate coefficients of x^n for $n>0$ and then rearrange to avoid subtractions. Interpret the rearranged result combinatorially. Can you give a direct proof of it?

(c) Let $q_{n,k}$ (resp. $p_{n,k}$) be the partitions counted in q_n (resp. p_n) in which no part exceeds k. Obtain formulas for $\sum_{n\geq 0}q_{n,k}x^n$ and $\sum_{n\geq 0}p_{n,k}x^n$.

10.4.5. Let a "pile" be, roughly, a two dimensional stack of square blocks resting on a flat surface, with each block directly on top of another and each row not having gaps. A more formal definition of a pile of height h is a sequence of $2h$ integers such that

$$0 = a_1 \le a_2 \le \cdots \le a_h < b_h \le \cdots \le b_2 \le b_1.$$

Here a block has width 1 and, counting rows with the first row on the bottom, the left end of row i is at a_i and the right end is at b_i. The number of blocks in the ith row is $b_i - a_i$ and the total number of blocks in the pile is $\sum_{i=1}^{h}(b_i - a_i)$. Let s_n be the number of n-block piles and $s_{n,h}$ the number of those of height h. Obtain a formula for $\sum s_n x^n$ and $\sum_{n\ge 0} s_{n,h} x^n$.
Hint. The generating function for partitions with no part exceeding k will be useful.

10.4.6. Let $a_1 < a_2 < \cdots < a_k$ be a k element subset of $\underline{n} = \{1, 2, \ldots, n\}$. We will study subsets with restrictions on the a_i.

(a) Let $a_0 = 0$. By looking at $a_i - a_{i-1}$, show that there is a bijection between k element subsets of $\underline{n}$ and k long sequences of positive integers with sum not exceeding n.

(b) Let u_n be the number of k element subsets of $\underline{n}$. Use (a) to show that

$$U(x) = \Big(\sum_{i\ge 1} x^i\Big)^k \Big(\sum_{i\ge 0} x^i\Big) = \frac{x^k}{(1-x)^{k+1}}.$$

(Do *not* use the fact that $\binom{-k-1}{n-k} = \binom{n}{n-k}$.)

(c) Let t_n be the number of k element subsets $a_1 < a_2 < \cdots$ of $\underline{n}$ such that i and a_i have the same parity. In other words a_{2j} is even and a_{2j+1} is odd. Show that

$$T(x) = \frac{x^k}{(1-x^2)^k}\,\frac{1}{1-x} = \frac{(1+x)x^k}{(1-x^2)^{k+1}}.$$

(d) Let $\lfloor x \rfloor$ be the result of rounding x down; e.g., $\lfloor 3.5 \rfloor = 3$. Show that $t_n = \binom{\lfloor (n+k)/2 \rfloor}{k}$.

(e) We call (a_i, a_{i+1}) a succession if they differ by one. Let $s_{n,j}$ be the number of k element subsets of $\underline{n}$ with exactly j successions. Show that

$$S(x,y) \;=\; \frac{1}{1-x}\,\frac{x}{1-x}\,(xy + x^2 + x^3 + \cdots)^{k-1} \;=\; \frac{x^k(x + y(1-x))^{k-1}}{(1-x)^{k+1}}.$$

(f) Show that $\sum_{n\ge 0} s_{n,j} x^n \;=\; \binom{k-1}{j} x^{2k-j-1}(1-x)^{-(k+1-j)}$.

(g) Express $s_{n,j}$ as a product of two binomial coefficients. Check your result by listing all 4 element subsets of $\{1, \ldots, 6\}$ and determining how many successions they have.

10.4.7. Recall that a binary RP-tree to be an RP-tree where each vertex may have at most two sons. The set $\mathcal{T}$ of such trees was studied in Exercise 10.2.6, where we counted them by number of vertices.

(a) Using the Rules of Sum and Product, derive the relation $T(x) = x + xT(x) + xT(x)^2$ that led to

$$T(x) \;=\; \frac{1 - x - \sqrt{1 - 2x - 3x^2}}{2x}$$

in Exercise 10.2.6

(b) Discuss how you might compute the number of such trees. In particular, can you find a simple expression as a function of n?

10.4.8. Change the definition in Exercise 10.4.7 so that, if a node has just one son, then we distinguish whether or not it is a right or a left son. (This somewhat strange sounding distinction is sometimes important.) How many such trees are there with n internal vertices?

10.4.9. A rooted tree will be called "contractible" if it has a vertex with just one son since one can imagine combining that vertex's information with the information at its son.

(a) Find the generating function for the number of unlabeled noncontractible RP-trees, counting them by number of vertices.

(b) Find the generating function for the number of unlabeled noncontractible RP-trees, counting them by number of leaves.

(c) Obtain a linear differential equation with polynomial coefficients and thence a recursion from each of the generating functions in this problem.
Hint. Solve for the square root, differentiate, multiply by the square of the square root and then replace the square root that remains.

10.4.10. Let $t_{n,k}$ be the number of RP-trees with n leaves and k internal vertices (i.e., nonleaves).

(a) Find a generating function for $T(x, y)$.

(b) Using the previous result, prove that $t_{n,k} = t_{k,n}$ when $n + k > 1$.
Hint. Compare $T(x, y)$ and $T(y, x)$.

*(c) Find a bijection that proves $t_{n,k} = t_{k,n}$ when $n + k > 1$; that is, find a map from RP-trees to RP-trees that carries leaves to internal vertices and vice versa for trees with more than one vertex. Write out your bijection for RP-trees with 5 vertices.
Hint. Describe the construction recursively (or locally).

10.4.11. Let D be a set of nonnegative integers such that $0 \in D$. For this exercise, we'll say that an RP-tree is of outdegree D if the number of sons of each vertex lies in D. Thus, full binary RP-trees are of outdegree $\{0, 2\}$.

(a) Let $T_D(x)$ be the generating function for unlabeled RP-trees of outdegree D by number of vertices. Prove that

$$T_D(x) \;=\; x \sum_{d \in D} T_D(x)^d$$

(b) Show that the previous formula allows us to compute $T_D(x)$ recursively.

(c) Let $L_D(x)$ be the generating function for unlabeled RP-trees of outdegree D by number of leaves. Show that it doesn't make sense to talk about $L_D(x)$ when $1 \in D$, that

$$L_D(x) \;=\; \sum_{d \in D} L_D(x)^d - 1 + x,$$

and that this allows us to compute $L_D(x)$ recursively when $1 \notin D$.

10.4.12. We have boxes labeled with pairs of numbers like $(2, 6)$. The labels have the form (i, j) for $1 \le i \le 3$ and $1 \le j \le k$. Thus we have $3k$ boxes. Unlabeled balls are placed into the boxes. This must be done so that the number of balls in box (i, j) is a multiple of i and, for each j, the total number of balls in boxes $(1, j)$, $(2, j)$ and $(3, j)$ is at most 5. What is the generating function for the number of ways to place n balls?
Hint. Find the generating function for placing balls into $(1, *)$, $(2, *)$ and $(3, *)$ and then use the Rule of Product.

*10.4.13. An unlabeled full binary rooted tree is like the ordered (i.e., plane) situation except that we make no distinction between left and right sons. Let β_n be the number of such trees with n leaves and let $B(x) = \sum \beta_n x^n$. Show that $B(x) = x + (B(x)^2 + B(x^2))/2$.

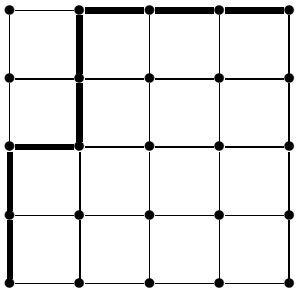


Figure 10.2 A path for $n = 4$ for Exercise 10.4.14.

10.4.14. Imagine the plane with lines like a sheet of graph paper; i.e., the lines $x = k$ and the lines $y = k$ are drawn in for all integers k. Think of the intersections of the lines vertices and the line segments connecting them as edges. The portion of the plane with $0 \le x \le n$ and $0 \le y \le n$ is then a simple graph with $(n+1)^2$ vertices. Let a_n be the number of paths from the vertex $(0,0)$ to the vertex (n,n) that never go down or left and that remain above the line $x = y$ except at $(0,0)$ and (n,n). Figure10.2 shows such a path for $n = 4$. We could describe such a path more formally as a sequence (x_i, y_i) of pairs of nonnegative integers such that $x_0 = y_0 = 0$, $x_{2n} = y_{2n} = n$, $x_i < y_i$ for $0 < i < 2n$ and $(x_i, y_i) - (x_{i-1}, y_{i-1})$ equals either $(1,0)$ or $(0,1)$. Draw some pictures to see what this looks like.

(a) Show that a_n is the number of sequences $s_1, \ldots, s_{2n}$ containing exactly n -1's and n 1's such that $s_1 + \cdots + s_k > 0$ for $0 < k < 2n$.

(b) By looking at $s_2, \ldots, s_{2n-1}$ for $n > 1$, conclude that $A(x) = x + \sum_{k \ge 1} x\, A(x)^k$.

(c) Determine s_n. Note that this number is the same as the number of unlabeled full binary RP-trees with n leaves, which is the same as the number of unlabeled RP-trees with n vertices.

*(d) In the previous part, you concluded that the set $\mathcal{S}_n$ of paths of a certain type from $(0,0)$ to (n,n) has the same size as the set $\mathcal{T}_n$ of unlabeled RP-trees with n vertices. Find a bijection $f_n: \mathcal{S}_n \to \mathcal{T}_n$, and thus prove this equality without the use of generating functions.

10.4.15. Fix a set S of size s. For $n \ge 1$, let $a_{n,k}$ be the number of n long ordered lists that can be made using S so that we never have more than k consecutive identical entries in the list. Thus, with $k \ge n$ there is no restraint while with $k = 1$ adjacent entries must be different. Let $A_k(x) = \sum_{n \ge 0} a_{n,k} x^n$. There are various approaches to $A_k(x)$.

(a) By considering the last run of identical entries in the list and using the Rules of Sum and Product, show that

$$A_k(x) \;=\; s(x + x^2 + \cdots + x^k) + A_k(x)(s-1)(x + x^2 + \cdots + x^k).$$

(b) Find an explicit formula for $A_k(x)$.

(c) Show that $a_{n+1,k} \;=\; s a_{n,k} - (s-1) a_{n-k,k}$ for $n > k$ by using the generating function.

(d) Derive the previous recursion by a direct argument.

10.4.16. We claim that the set of sequences of zeroes and ones that do not contain either 101 or 111 is described by

$$0^* \cup \Big(0^*(1 \cup 11)\big(000^*(1 \cup 11)\big)^* 0^*\Big)$$

and each such sequence has a unique description. You need not verify this. Let a_n be the number of such sequences of length n and let $A(x)$ be the generating function for a_n.

(a) Derive the formula $A(x) = \frac{1+x+2x^2+x^3}{1-x-x^3-x^4}$.

(b) Using $A(x)$, obtain the recursion $a_n = a_{n-1} + a_{n-3} + a_{n-4}$ for $n \geq 4$ and find the initial conditions.

(c) Using $1 - x - x^3 - x^4 = (1 - x - x^2)(1 + x^2)$, derive the formula

$$a_n = \frac{7F_{n+1} + 4F_n - b_n}{5} \quad \text{where} \quad b_n = \begin{cases} 2(-1)^{n/2}, & \text{if } n \text{ is even,} \\ (-1)^{(n-1)/2}, & \text{if } n \text{ is odd,} \end{cases}$$

where the Fibonacci numbers are given by $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$; that is, their generating function is $\frac{x}{1-x-x^2}$.

(d) Prove that

$$a_{2n} = F_{n+2}^2 \quad \text{and} \quad a_{2n+1} = F_{n+2}F_{n+3} \quad \text{for } n \geq 0.$$

Hint. Show that the recursion and initial conditions are satisfied.

10.4.17. Using partial fractions, obtain a formula for r_n from (10.33).

*10.4.18. Let G be the simple graph with vertex set $\underline{n} \cup \{0\}$ and the $2n$ edges $\{n, 1\}$, $\{i, i+1\}$ $(1 \leq i < n)$ and $\{0, j\}$ $(1 \leq j \leq n)$, except for $n = 1, 2$ where we must avoid adding $\{n, 1\}$ in order to get a simple graph. In other words, G is like the graph in Example 10.17 except that one more edge $\{1, n\}$ has been added so that the picture looks like a wheel with spokes for $n > 2$. We want to know how many spanning trees G has.

(a) Let $\mathcal{T}$ be as in Example 10.17 and let $\mathcal{T}'$ consist of the trees in $\mathcal{T}$ with one of the nonroot vertices marked. Choose one tree from $\mathcal{T}'$ and then a, possibly empty, sequence of trees from $\mathcal{T}$. Suppose we have a total of n nonroot vertices. Merge the root vertices and relabel the nonroot vertices 1 to n, starting with the marked vertex in the tree from $\mathcal{T}'$ and preceding cyclically until all nonroot vertices have been labeled. Explain why this gives all the spanning trees exactly once.

(b) Show that $\mathsf{G}_{\mathcal{T}'}(x) = x(d/dx)(\mathsf{G}_{\mathcal{T}}(x)) = x(1+x)/(1-x)^3$.

(c) Show that generating function for the spanning trees is

$$\frac{x(1+x)}{(1-x)(1-3x+x^2)}.$$

(d) Show that the number of spanning trees is $2r_{n+1} - 3r_n - 2$, where r_n is given in Example 10.17.

10.4.19. We define a set of *regular sequences* (or regular strings) on the "alphabet" A. (An alphabet is any finite set.) Let R, R_1, and R_2 stand for sets of regular strings on A. The sets of regular strings on A are the empty set, the sets $\{a\}$ where $a \in A$, and the sets that can be built recursively using the following operations:

- union ("or") of sets, i.e., the set of all strings that belong to either R_1 or R_2;
- juxtaposition ("and then"), i.e., the set of all strings $r_1 r_2$ where $r_1 \in R_1$ and $r_2 \in R_2$;
- arbitrary iteration R^*, i.e., for all $n \geq 0$, all strings of the form $r_1 r_2 \dots r_n$ where $r_i \in R$. (The empty string is obtained when $n = 0$.)

See Example 10.16 for a specific example of a set of regular sequences The purpose of this exercise is to construct a nondeterministic finite automaton that recognizes any given set of regular strings. Nondeterministic finite automata are defined in Section 6.6 (p. 189). We will build up the machine by following the way the strings are built up.

(a) Let $\mathcal{A}$ be an automaton. Show that there is another automaton $S(\mathcal{A})$ that recognizes the same strings and has no edges leading into the start state.
Hint. Create a new state, let it be the start state and let it have edges to all of the states the old start state did. Remember to specify the accepting states, too.

(b) If $\mathcal{A}$ recognizes the set A and $\mathcal{B}$ recognizes the set B, construct and automaton that recognizes the set $A \cup B$.
Hint. Adjust the idea in (a).

(c) If $\mathcal{A}$ recognizes A, construct an automaton that recognizes A^*.
Hint. Add some edges.

(d) If $\mathcal{A}$ recognizes the set A and $\mathcal{B}$ recognizes the set B, construct and automaton that recognizes AB; i.e., the set $A \times B$.

Notes and References

The classic books on generating functions are those by MacMahon [6] and Riordan [7]. They are quite difficult reading and do not take a "combinatorial" approach to generating functions. There are various combinatorial approaches. Some can be found in the texts by Wilf [10] and Stanley [9, Ch. 3] and in the articles by Bender and Goldman [1] and Joyal [5]. The articles are rather technical.

Parts of the texts by Greene and Knuth [4] and by Graham, Knuth and Patashnik [3] are oriented toward computer science uses of generating functions. See also the somewhat more advanced text by Sedgewick and Flajolet [7]. Wilf [10] gives a nice introduction to generating functions. Goulden and Jackson's book [2] contains a wealth of material on generating functions, but is at a higher level than our text.

We have studied only the simplest sorts of recursions. Recursions that require more sophisticated methods are common as are recursions that cannot be solved exactly. Sometimes approximate solutions are possible. We don't know of any systematic exposition of techniques for such problems.

We have not dealt with the problem of defining formal power series; that is, defining a generating function so that the convergence of the infinite series is irrelevant. An introduction to this can be found in the first few pages of Stanley's text [9].

1. Edward A. Bender and Jay R. Goldman, Enumerative uses of generating functions, *Indiana Univ. Math. J.* **20** (1971), 753–765.
2. Ian P. Goulden and David M. Jackson, *Combinatorial Enumeration*, Dover (2004).
3. Ronald L. Graham, Donald E. Knuth and Oren Patashnik, *Concrete Mathematics*, 2nd ed., Addison-Wesley, Reading (1994).
4. Daniel H. Greene and Donald E. Knuth, *Mathematics for the Analysis of Algorithms*, 3rd ed., Birkhäuser (1990).

5. André Joyal, Une théorie combinatoire des séries formelles, *Advances in Math.* **42** (1981), 1-82.
6. Percy Alexander MacMahon, *Combinatory Analysis*, Chelsea, New York, 1960. Reprint of two volume Cambridge Univ. Press edition (1915, 1916).
7. John Riordan, *An Introduction to Combinatorial Analysis*, Princeton Univ. Press (1980).
8. Robert Sedgewick and Philippe Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley (1996).
9. Richard P. Stanley, *Enumerative Combinatorics*, vols. 1 and 2. Cambridge Univ. Press (1999, 2001).
10. Herbert S. Wilf, *Generatingfunctionology*, 2nd ed., Academic Press (1993).

CHAPTER 11

*Generating Function Topics

Introduction

The four sections in this chapter deal with four distinct topics: systems of recursions, exponential generating functions, Pólya's Theorem and asymptotics estimates. These sections can be read independently of one another. The section on "asymptotic" estimates refers to formulas in earlier sections of the chapter, but there is no need to read the section containing the formula.

"Systems of recursions," as you might guess, deals with the creation and solution of sets of simultaneous recursions. These can arise in a variety of ways. One source of them is algorithms that involve interrelated recursive subroutines. Another source is situations that can be associated with grammars. General context free grammars can lead to complicated recursions, but regular grammars (which are equivalent to finite state automata) lead to simple systems of recursions. We limit our attention to these simple systems.

Exponential generating functions are very much like ordinary generating functions. They are used in situations where things are labeled rather than unlabeled. For example, they are used to study partitions of a set because each of the elements in the set is different and hence can be thought of as a labeled object. We will briefly study the Rules of Sum and Product for them as well as a useful consequence of the latter—the Exponential Formula.

Burnside's Lemma (Theorem 4.5 (p. 112)) is easily seen to apply to the situation in which the objects being counted have "weights". As a result, we can introduce generating functions into the study of objects with symmetries. This "weighted Burnside lemma" has a variety of important special cases. We will study the simplest, and probably most important one—Pólya's theorem.

Suppose we are studying some sequence of numbers a_n and want to know how the sequence behaves when n is large. Usually it grows rapidly, but we want to know more than that—we want a relatively simple formula that provides some sort of estimate for a_n. Stirling's formula (Theorem 1.5 (p. 12)) is an example of such a formula. This subject is referred to as *asymptotics* or *asymptotic analysis*. Because the proofs of results in this area require either messy estimates, a knowledge of complex variables or both, we will not actually prove the estimates that we derive.

11.1 Systems of Recursions

So far we've dealt with only one recursion at a time. Now we look at ways in which systems of recursions arise and adapt our methods for a single recursion to solving simple systems of recursions. The adaptation is straightforward—allow there to be more than one recursion. As usual, examples are the best way to see what this is all about.

Example 11.1 Counting Batcher sort comparators Let's study the Batcher sort. As with our study of merge sorting in Example 10.4 (p. 278), we'll limit the number of things being sorted to a power of 2 for simplicity. We want to determine b_k, the number of comparators in a Batcher sort for 2^k things. We'll rewrite the Batcher sorting algorithm in Section 8.3.2 to focus on the number of things being sorted and number of comparators. The comments indicate the contributions to the recursion.

```
BSORT(2^k things)                          /* uses b_k comparators */
     If k = 0, Return                      /* b_0 = 0 */
     BSORT(2^(k-1) things)                 /* b_k = b_(k-1) */
     BSORT(2^(k-1) things)                 /*       +b_(k-1) */
     BMERGE(2^k things)                    /*       +m_k */
     Return
End

BMERGE(2^k things)                         /* uses m_k comparators */
     If k = 0, Return                      /* m_0 = 0 */
     End if
     If k = 1,
          one Comparator and Return        /* m_1 = 1 */
     BMERGE2(2^k things)                   /* m_k = t_k */
     2^(k-1) - 1 Comparators               /*       +2^(k-1) - 1 */
     Return
End

BMERGE2(2^k things)                        /* uses t_k comparators */
     BMERGE(2^(k-1) things)                /* t_k = m_(k-1) */
     BMERGE(2^(k-1) things)                /*       +m_(k-1) */
     Return
End
```

Note that since `BMERGE2(`2^k `things)` is never called for $k < 2$, we can define t_0 and t_1 arbitrarily.

How should we choose the values of t_0 and t_1? In the end, it doesn't really matter because the answers will be unaffected by our choices. On the other hand, how we choose these values can affect the amount of work we have to do. There are two rules of thumb that can help in making such choices:

- Choose the initial values so as to minimize the number of special cases of the recursion. (For example, the recursion below has two special cases for m_k.)
- Choose simple values like zero.

We'll set $t_0 = 0$ and $t_1 = 2m_0 = 0$.

Copying the recursions from the comments in the code we have

$$\begin{aligned} b_k &= \begin{cases} 0 & \text{if } k = 0, \\ 2b_{k-1} + m_k & \text{if } k > 0; \end{cases} \\ m_k &= \begin{cases} 0 & \text{if } k = 0, \\ 1 & \text{if } k = 1, \\ t_k + 2^{k-1} - 1 & \text{if } k > 1; \end{cases} \\ t_k &= \begin{cases} 0 & \text{if } k = 0, \\ 2m_{k-1} & \text{if } k > 0. \end{cases} \end{aligned} \tag{11.1}$$

We now apply our six steps for solving recursions (p. 277), allowing more than one recursion and more than one resulting equation. Let $B(x)$, $M(x)$ and $T(x)$ be the generating functions. The result of applying the first four steps to (11.1) is

$$\begin{aligned} B(x) &= 2xB(x) + M(x), \\ M(x) &= x + \sum_{k\geq 2} t_k x^k + \sum_{k\geq 2} (2^{k-1} - 1)x^k \\ &= x + \sum_{k\geq 0} t_k x^k + \sum_{k\geq 1} (2^{k-1} - 1)x^k \\ &= x + T(x) + \frac{x}{1-2x} - \frac{x}{1-x}, \\ T(x) &= 2xM(x). \end{aligned}$$

We now carry out Step 5 by solving these three linear equations in the three unknowns $B(x)$, $M(x)$ and $T(x)$. From the first equation, we have

$$B(x) = \frac{M(x)}{1-2x}. \tag{11.2}$$

Combining the equations for $M(x)$ and $T(x)$, we have

$$M(x) = x + 2xM(x) + \frac{x}{1-2x} - \frac{x}{1-x}.$$

Solving this for $M(x)$ and substituting the result (11.2), we obtain

$$B(x) = \frac{x}{(1-2x)^2} + \frac{x}{(1-2x)^3} - \frac{x}{(1-x)(1-2x)^2}.$$

Our formula for $B(x)$ can be expanded using partial fractions. We won't carry out the calculations here except for noting that we can rewrite this as

$$B(x) = \frac{x}{(1-2x)^3} - \frac{x}{(1-2x)^2} + \frac{2x}{1-2x} - \frac{2x}{1-x}.$$

The result is

$$b_k = 2^{k-2}(k^2 - k + 4) - 1. \tag{11.3}$$

How does this result compare with the upper bound on the number of comparisons in merge sort that we found in Example 10.4? There we obtained an upper bound of $n \log_2 n$ and here we obtained an actual value that is close to $\frac{1}{2}n(\log_2 n)^2$. Thus Batcher sort is a poor software sorting algorithm. On the other hand, it is a much better network sorting algorithm than the only other one we studied, the brick wall. □

In the next example we will work with a set of linked recursions in the context of a directed graph or, equivalently, a finite state automaton.

Figure 11.1 Three ways to place nonoverlapping dominoes on 2 by 5 boards. A domino is indicated by a white rectangle and the board is black.

Example 11.2 Rectangular arrays, digraphs and finite automata

Imagine a k by n array of squares. We would like to place dominoes on the array so that each domino covers exactly two squares and the dominoes do not overlap. Not all squares need be covered. Thus there may be any number of dominoes from none to $nk/2$. Let $a_n(k)$ be the number of ways to do this. Figure 11.1 shows some ways to place dominoes on a 2 by 5 board.

We can work out a recursion for $a_n(1)$ quite easily. Let's do it. When $n > 1$, the number of arrangements with the last square empty is $a_{n-1}(1)$ and the number of arrangements with a domino in the last square is $a_{n-2}(1)$. This is the same as the recursion for the Fibonacci numbers, but the initial conditions are a bit different. (What are they?) After some calculation, you should be able to get $a_n(1) = F_{n-1}$.

There is a quicker way to get $a_n(1)$. If we replace each empty square with a "0" and each domino with "10", we obtain an n-long string of zeroes and ones ending in zero and having no adjacent ones. Conversely any such string of zeroes and ones gives rise to a placement of dominoes. By stripping off the rightmost zero in the string, we have that $a_n(1) = F_{n-1}$.

For $k > 1$, $a_n(k)$ is much more complicated. Try to write down a recursion for $a_n(2)$—or calculate it in any other manner.

* * * Stop and think about this! * * *

We will show how to use a directed graph to produce our patterns of dominoes. The graph can be regarded as a finite state machine or, more specifically, a nondeterministic finite automaton (Section 6.6). We will show how to associate generating functions with such digraphs.

Our machine (digraph) has four states (vertices), which we call *clear, first, second* and *both*. We imagine moving across the $2 \times n$ array, one column at a time from left to right, placing dominoes as we reach a column. When we decide to place a domino in a horizontal position starting in column j of the array, we are covering a square in column $j+1$ that we haven't yet reached. We call this covering of an unreached square a "commitment." At the next step we must take into account any commitment that was made in the previous step. Our states keep track of the unsatisfied commitments; that is, if we are filling the jth column, the commitments made at column $j-1$:

(a) *clear* means there are no commitments to take into account;

(b) *first* means we made a commitment by placing a domino in the first row of column $j-1$ which runs over into column j;

(c) *second* means we made a commitment by placing a domino in the second row;

(d) *both* means we made commitments by placing dominoes in both rows.

Using the letters, c, f, s and b, the sequences of states for the columns in Figure 11.1 are $fsfcc$, $fcfsc$ and $ccscc$, respectively. The columns to which the commitments are made are 2 through 6. No commitments are made to column 1 because no dominoes are placed before column 1. If we wanted to reflect this fact in our sequences, we could add an entry of c for column 1 at the beginning of each sequence. Note that all of these strings end with a c because no dominoes hang over the right end of the board.

There is an edge in our digraph from state x to state y for each possible way to get from state x at column j of the array to state y at column $j+1$ of the array. For example, we can go from *clear* to *clear* by placing no dominoes in column j or by placing one vertical domino in the column. Thus there are two loops at *clear*. Since we cannot go from *first* to *both*, there are no edges from

first to *both*. The following table summarizes the possibilities. An entry $m_{x,y}$ in position (x, y) is the number of edges from state x to state y.

	clear	*first*	*second*	*both*
clear	2	1	1	1
first	1	0	1	0
second	1	1	0	0
both	1	0	0	0

To complete our picture, we need the initial and accepting states. From the discussion at the end of the previous paragraph, you should be able to see that our initial state should be *clear* and that there should be one accepting state, which is also *clear*.

Let c_n be the number of ways to reach the state *clear* after taking n steps from the starting state, *clear*. This means that no dominoes hang over into column $n+1$. Note that c_n is the number of ways to place dominoes on a $2 \times n$ board. Define f_n, s_n and b_n in a similar way according to the state reached after n steps from the starting state. Let $C(x)$, etc., be the corresponding generating functions.

We are only interested in c_n (or $C(x)$), so why introduce all these other functions? The edges that lead into a state give us a recursion for that state, for example, looking down the column labeled *clear*, we see that

$$c_{n+1} = 2c_n + f_n + s_n + b_n \tag{11.4}$$

for $n \geq 0$. Thus, when we study c_n this way, we end up needing to look at the functions f_n, s_n and b_n, too.

In a similar manner to the derivation of (11.4),

$$\begin{aligned} f_{n+1} &= c_n + s_n, \\ s_{n+1} &= c_n + f_n, \\ b_{n+1} &= c_n, \end{aligned} \tag{11.5}$$

for $n \geq 0$. The initial conditions for the recursions (11.4) and (11.5) are the values at $n = 0$. Since the initial state, *clear*, is the only state we can get to in zero steps,

$$c_0 = 1 \quad \text{and} \quad f_0 = s_0 = b_0 = 0.$$

To see how much work these recursions involve, use them to find c_8, the number of ways to place dominoes on a 2 by 8 board. Can we find an easier way to calculate c_n; for example, one that does not require the values of f, s and d as well? Yes.

We begin by converting the recursions into a set of linked generating functions using our method for attempting to solve recursions. Multiplying both sides of the equations (11.4) and (11.5) by x^{n+1}, summing over $n \geq 0$ and using the initial conditions, we obtain

$$\begin{aligned} C(x) &= x(2C(x) + F(x) + S(x) + B(x)) + 1 \\ F(x) &= x(C(x) + S(x)) \\ S(x) &= x(C(x) + F(x)) \\ B(x) &= xC(x). \end{aligned}$$

We have four linear equations in the four unknowns $C(x)$, $F(x)$, $S(x)$ and $B(x)$. Let's solve them for $C(x)$. Subtracting the second and third equations, we get $F(x) - S(x) = x(S(x) - F(x))$ for all x and so $S(x) = F(x)$. Thus $F(x) = xC(x) + xF(x)$ and so $F(x) = S(x) = xC(x)/(1-x)$. Substituting this and $B(x) = xC(x)$ into the first equation gives us

$$C(x) = 2xC(x) + \frac{2x^2C(x)}{1-x} + x^2C(x) + 1.$$

With a bit of algebra, we easily obtain

$$C(x) = \frac{1-x}{1-3x-x^2+x^3}. \tag{11.6}$$

A method of obtaining this by working directly with the table, thought of as a matrix

$$M = \begin{pmatrix} 2 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{11.7}$$

is given in Exercises 11.1.7.

We can use partial fractions to determine c_n, but before doing so, we'd like to note that (11.6) gives us an easier recursion for calculating c_n: From (11.6) we have $C(x) = 1-x+(3x+x^2-x^3)C(x)$. Looking at the coefficient of x^n on both sides, we have

$$c_0 = 1, \qquad c_1 = -1 + 3c_0 = 2, \qquad c_2 = 3c_1 + c_0 = 7$$

and

$$c_n = 3c_{n-1} + c_{n-2} - c_{n-3} \quad \text{for} \quad n > 2. \tag{11.8}$$

Use this recursion to find c_8 and compare the amount of work with that when you were using (11.4) and (11.5). It's also possible to derive (11.8) by manipulating (11.4) and (11.5) without using generating functions. (Try doing it.)

To use partial fractions, we must factor the denominator of (11.6):

$$1 - 3x - x^2 + x^3 = (1-px)(1-qx)(1-rx).$$

Unfortunately, the cubic does not factor with rational p, q or r. Using numerical methods we found that $p = 3.21432\ldots$, $q = .46081\ldots$ and $r = -.67513\ldots$. By partial fractions, $c_n = Pp^n + Qq^n + Rr^n$, where $P = .66459\ldots$, $Q = .07943\ldots$ and $R = .25597\ldots$. Thus a_n is the integer closest to Pp^n. Since we know p and P only approximately, we can't get c_n exactly for large n this way. Instead, we'd use the recursion (11.8) to get exact values for c_n. On the other hand, the recursion gives us no idea about how fast c_n grows, so we'd use our partial fraction result for this sort of information. For example, the result says that, in some sense, the average number of choices per column is about p since if there were exactly p choices per column, there would be p^n ways to place dominoes in n columns. □

Example 11.3 Binary operations Let $\wedge$ denote exponentiation; e.g., $3 \wedge 2 = 9$. The interpretation of $2 \wedge 3 \wedge 2$ is ambiguous. We can remove the ambiguity by using parentheses:

$$(2 \wedge 3) \wedge 2 = 8 \wedge 2 = 64$$
$$2 \wedge (3 \wedge 2) = 2 \wedge 9 = 128.$$

Sometimes we get the same answer from different parenthesizations; e.g., $2 \wedge 2 \wedge 2 = 16$, regardless of parentheses.

Let's consider the possible values of

$$0 \wedge 0 \wedge \ldots \wedge 0. \tag{11.9}$$

Unfortunately, $0 \wedge 0$ is not defined for the real numbers. For the purpose of this example, we'll define $0 \wedge 0 = 1$. The only other powers we need are well defined. In summary

$$0 \wedge 0 = 1, \quad 0 \wedge 1 = 0, \quad 1 \wedge 0 = 1 \quad \text{and} \quad 1 \wedge 1 = 1. \tag{11.10}$$

You should be able to show by induction on the length of the string that the only possible values for (11.9) are 0 and 1, regardless of how the expression is parenthesized.

How many of the parenthesizations of (11.9) give the expression the value 0 and how many give it the value 1?

If there are n zeroes present, let z_n be the number of ways to obtain the value 0 and let w_n be the number of ways to obtain 1. We can write down the generating functions by using the Rules of

Sum and Product: A zero is produced if $n = 1$ OR, by (11.10), if the left string is a zero AND the right string is a one. By (11.10), the other three combinations give a one. Thus

$$\begin{aligned} Z(x) &= x + Z(x)W(x) \\ W(x) &= Z(x)Z(x) + W(x)Z(x) + W(x)W(x). \end{aligned} \tag{11.11}$$

These equations are more complicated than our earlier ones because they are not linear. In general, we cannot hope to solve such equations; however, these can be solved. Try to do so before you continue.

* * * Stop and think about this! * * *

Let $T(x) = W(x) + Z(x)$, the total number of parenthesizations without regard to value. Using algebra on (11.11) or, more simply, using a direct combinatorial argument, you should be able to show that

$$T(x) = x + T(x)T(x).$$

The solution to this equation is

$$T(x) = \frac{1 - \sqrt{1 - 4x}}{2}, \tag{11.12}$$

where the minus sign was chosen on the square root because $T(0) = t_0 = 0$. We now rewrite $Z(x) = x + Z(x)W(x)$ as

$$Z(x) = x + Z(x)\big(T(x) - Z(x)\big) = x + T(x)Z(x) - Z(x)^2. \tag{11.13}$$

Solving this quadratic for $Z(x)$:

$$Z(x) = \frac{T(x) - 1 + \sqrt{\big(1 - T(x)\big)^2 + 4x}}{2}, \tag{11.14}$$

where the root with the plus sign was chosen because $Z(0) = 0$.

Of course, we can substitute (11.12) into (11.14) to obtain the explicit formula for $Z(x)$. Unfortunately, we are unable to extract a nice formula for z_n from the result.

What can be done? At present, not much; however, in Example 11.31 (p. 351), we will obtain estimates of z_n for large n.

Note that $T(x) = B(x)$, the generating function for unlabeled full binary RP-trees. We obtained a differential equation for B in Example 10.8 (p. 284). This led to a simple recursion. There are general techniques for obtaining such differential equations and hence recursions, but they are beyond the scope of this text. We merely remark that it is possible to obtain a recursion for z_n that requires much less work than would be involved in simply using the recursion that is obtained by extracting the coefficient of x^n in (11.13). □

Exercises

*11.1.1. Derive (11.8) directly from (11.4) and (11.5) without using generating functions.

11.1.2. Redo Exercise 10.2.4 (p. 280) using the directed graph method of Example 11.2. Which way was easier?
Hint. Here's one way to set it up. Introduce three vertices to indicate the present digit: *zero*, *one* and *two*. You can introduce a bad vertex, too, or make certain digits impossible. Since you can end at any digit, you'll want to add generating functions together to get your answer.

11.1.3. Let a_n be the number of ways to place dominoes on a 3 by n board with no blank spaces. Note that $a_n = 0$ when n is odd. Let $A(x)$ be the generating function.

(a) Prove that $A(x) = (1 - x^2)/(1 - 4x^2 + x^4)$.

(b) Show that $a_n = 4a_{n-2} - a_{n-4}$ when $n \geq 4$.

*(c) Prove the previous recursion without the use of generating functions.

11.1.4. How many n-long sequences of zeroes ones and twos are there with no adjacent entries equal?

11.1.5. Let $\mathcal{L}$ be a set of strings and let a_n be the number of strings in $\mathcal{L}$ of length n. Suppose that we are given a nondeterministic finite automaton for recognizing precisely the strings in $\mathcal{L}$. Describe a method for using the automaton to get the generating function for the a_n's.

11.1.6. Let $a_{n,j}(2)$ be the number of ways to place exactly j dominoes on a 2 by n board. Extend the finite machine approach of Example 11.2 so that you can calculate $\sum_{n,j} a_{n,j}(2)x^n y^j$.
Hint. Put on the edges of the digraph information about the number of dominoes added to the board. Use it to write down the equations relating the generating functions.

11.1.7. Given a finite machine as in Example 11.2, let $m_{x,y}$ be the number of edges from state x to y. This defines a matrix M of (11.7).

(a) Show that $m^{(n)}_{x,y}$, the (x, y) entry in M^n, is the number of ways to get from x to y in exactly n steps.

(b) Let $\mathbf{a}$ be a row vector with $a_k = 1$ if k is an accepting state and $a_k = 0$ otherwise. Define $\mathbf{i}$ similarly for the initial state. Show that $\mathbf{i}M^n\mathbf{a}^t$ is the number of ways to get from the initial state to an accepting state in exactly n steps. (We use $\mathbf{a}^t$ for the column vector which is the transpose of the row vector $\mathbf{a}$.)

(c) Conclude that the generating functions in Example 11.2 and in Exercise 11.1.5 have the form $\mathbf{i}(I - xM)^{-1}\mathbf{a}^t$, where I is the identity matrix.

(d) Extend the definition of M to include the previous exercise.

11.1.8. Let a_n be the number of ways to distribute unlabeled balls into boxes numbered 1 through n such that for each j with $1 \leq j < n$, the number of balls in boxes j and $j + 1$ together is at most 2.

(a) Construct a directed graph that can be used to calculate the generating function for a_n.
Hint. Let the vertices (states) be the number of balls in a box.

(b) Obtain a set of linked equations for the various generating functions.

(c) Show that $\sum a_n x^n = (1 + x - x^2)/(1 - 2x - x^2 + x^3)$.

(d) Restate the solution in matrix terms using the previous exercise.

(e) Find the roots of $r^3 - 2r^2 - r + 1 = 0$ numerically and use partial fractions to determine the value of a_n.

(f) Let $b_{n,k}$ be the number of ways to distribute k balls into n boxes subject to our adjacency condition. Using the idea in Exercise 11.1.6, determine the generating function for $b_{n,k}$.

11.1.9. How one approaches a problem can be very important. Obviously, the wrong attack on a problem may result in no solution. Less obvious is the fact that one approach may involve much less work than another. Everyone sometimes fails to find the easiest approach. You can probably find examples of this in the way you've solved some homework problems. Here's another brief example. How many n long sequences of zeroes and ones contain the pattern $p = 010101011$?

(a) One approach is to use two of the tools we've developed: designing finite automata for accepting sequences with certain patterns and obtaining generating functions from automata. This would require an automaton with at least ten states. Draw such an automaton for p and write down the family of associated equations.

(b) We'll look at a simpler approach. Let a_n be the number of n long strings that do *not* contain the desired pattern p. By considering what happens when something is added to the end of an $n-1$ long pattern, show that $2a_{n-1} = a_n + c_n$ for $n > 0$, where c_n is the number of n long strings $s_1, \ldots, s_n$ containing p but with p not contained in $s_1, \ldots, s_{n-1}$. Also show that $c_n = 0$ for $n < 9$, the length of p.

(c) Show that $c_n = a_{n-9}$ and conclude that $A(x) = (1 - x + x^9)^{-1}$.
Hint. If your proof also works for 001001001, it is not quite correct.

(d) Generalize the previous result as much as you can.

(e) Show that the finite automata approach might sometimes be better by giving an automaton for recognizing the strings that contain the pattern 001001001.

11.1.10. The situation we studied in Example 11.3 can be generalized in various ways. In this exercise you will study some possibilities.

(a) Suppose we have a set A of symbols and a binary operation $\circ$ on A. This means that for all $s, t \in A$ the value of $s \circ t \in A$ is given. Consider the "product" $b \circ b \circ \ldots \circ b$. We want to know how many ways each of the elements $s \in A$ can arise by parenthesizing this product. Describe carefully how to obtain the equations relating the generating functions from the "multiplication" table for the operation $\circ$.

(b) We can change the previous situation by letting $\circ$ be a k-ary operation. For example, if it is a ternary operation, there is no way to make sense of either the expression $b \circ b$ or the expression $b \circ b \circ b \circ b$. On the other hand, we have three parenthesizations for the 5-long case:

$$(b \circ b \circ b) \circ b \circ b, \quad b \circ (b \circ b \circ b) \circ b \quad \text{and} \quad b \circ b \circ (b \circ b \circ b).$$

Again, describe how to construct equations relating the generating functions.

11.2 Exponential Generating Functions

When we use ordinary generating functions, the parts we are counting are "unlabeled." It may appear at first sight that this was not so in all the applications of ordinary generating functions. For instance, we had sequences of zeroes and ones. Isn't this labeling the positions in a sequence? No, it's dividing them into two classes. If they were labeled, we would require that each entry in the sequence be different; that is, *each label would be used just once*. Well, then, what about placing balls into labeled boxes? Yes the boxes are all different, but the parts we are counting in our generating functions are the *unlabeled* balls, not the boxes. The boxes simply help to form the structure.

In this section, we'll use exponential generating functions to count structures with labeled parts. What we've said is rather vague and may have left you confused. We need to be more precise, so we'll look at a particular example and then explain the general framework that it fits into.

Recall the problem of counting unlabeled full binary RP-trees by number of leaves. We said that any such tree could be thought of as either a single vertex OR an ordered pair of trees. Let's look at the construction of this ordered pair a bit more closely. If the final tree is to have n leaves, we first choose some number k of leaves and construct a tree with that many leaves, then we construct a tree with $n - k$ leaves as the second member of the ordered pair. Thus there is a three step procedure:

1. Determine the number of leaves for the first tree (and hence also the second), say k.
2. Construct the first tree so that it contains k leaves.
3. Construct the second tree so that it contains $n-k$ leaves.

Now let's look at what happens if the leaves are to be labeled; i.e., there is a bijection from the n leaves to some set N of n labels. (Usually we have $N = \underline{n}$, but this need not be so.) In this case, we must replace our first step by a somewhat more complicated step and modify the other two steps in an obvious manner:

1′. Determine a subset K of N which will become the labels of the leaves of the first tree.
2′. Construct the first tree so that its leaves use K for labels.
3′. Construct the second tree so that its leaves use $N-K$ for labels.

Note that the number of ways to carry out Steps 2′ and 3′ depend only on $|N|$ and $|K|$, not on the actual elements of N and K. This is crucial for the use of exponential generating functions. Because of this, it is convenient to split Step 1′ into two steps:

1′a. Determine the number of leaves for the first tree (and hence also the second), say k.
1′b. Determine a subset K of N with $|K| = k$ to be the leaves of the first tree.

Let b_n be the number of unlabeled full binary RP-trees with n leaves and let t_n be the number of such trees except that the leaves have been labeled using some set N with $|N| = n$. For $n > 1$, our unlabeled construction gives us $\sum_k b_k b_{n-k}$, where the summation comes from the first step, the b_k from the second and the b_{n-k} from the third. Similarly, for the labeled construction, we have

$$\sum_{k=0}^{n} \binom{n}{k} t_k t_{n-k} \quad \text{for } n > 1, \tag{11.15}$$

where now the summation comes from Step 1′a and the binomial coefficient from Step 1′b.

The initial condition $t_1 = 1$ (and, if we want, $t_0 = 0$) together with (11.15) gives us a recursion for t_n. We'd like to use generating functions to solve this recursion as we did for the unlabeled case. The crucial step for the unlabeled case was the observation that we were dealing with a convolution which then led to (10.18). If this approach is to work in the labeled case, we need to be able to view (11.15) as a convolution, too.

Recall that a convolution is something of the form $\sum_k a_k c_{n-k}$. Unfortunately, (11.15) contains $\binom{n}{k}$ which is not a function of k and is not a function of $n-k$, so it can't be included in a_k or in c_{n-k}. Fortunately, we can get around this by rewriting the binomial coefficient in terms of factorials:

$$t_n \;=\; \sum_{k=0}^{n} \binom{n}{k} t_k t_{n-k} \;=\; n! \sum_{k=0}^{n} \frac{t_k}{k!} \frac{t_{n-k}}{(n-k)!}.$$

If we divide this equation by $n!$, we get a recursion for $t_n/n!$ in which the sum is a convolution. Thus, the generating function $B(x) = \sum b_n x^n$ in the unlabeled case should be replaced by the generating function $T(x) = \sum (t_n/n!)x^n$ in the labeled case. We can then proceed to solve the problem just as we did in the unlabeled case.

You may have noticed that $t_n = b_n n!$, a result which can be proved directly. So why go through all this? We did it to introduce an idea, not to solve a particular problem. So let's formulate the idea.

Definition 11.1 Exponential generating function *The* **exponential generating function** *for the sequence c is*

$$\sum_{n \geq 0} c_n (x^n/n!).$$

If $\mathcal{T}$ is some set of structures and $w(T)$ is the number of labels in T, then $\mathsf{E}_{\mathcal{T}}(x)$, the exponential generating function for $\mathcal{T}$ is $\sum_{T \in \mathcal{T}} x^{w(T)}/(w(T))!$. We abbreviate "exponential generating function" to **EGF**.

Theorem 11.1 Rule of Sum *Suppose a set $\mathcal{T}$ of structures can be partitioned into sets $\mathcal{T}_1, \ldots, \mathcal{T}_j$ so that each structure in $\mathcal{T}$ appears in exactly one $\mathcal{T}_i$. It then follows that*

$$\mathsf{E}_{\mathcal{T}} \;=\; \mathsf{E}_{\mathcal{T}_1} + \cdots + \mathsf{E}_{\mathcal{T}_j}.$$

Theorem 11.2 Rule of Product *Suppose each structure in a set $\mathcal{T}$ of structures can be constructed from an ordered partition (K_1, K_2) of the labels and some pair (T_1, T_2) of structures using the labels K_1 in T_1 and K_2 in T_2 such that:*

(*i*) *The number of ways to choose a T_i with labels K_i depends only on i and $|K_i|$.*

(*ii*) *Each structure $T \in \mathcal{T}$ arises in exactly one way in this process.*

(We allow the possibility of $K_i = \emptyset$ if $\mathcal{T}_i$ contains structures with no labels.) It then follows that

$$\mathsf{E}_{\mathcal{T}}(x) \;=\; \mathsf{E}_1(x)\mathsf{E}_2(x),$$

where $\mathsf{E}_i(x) \;=\; \sum_{n=0}^{\infty} t_{i,n} x^n/n!$ and $t_{i,n}$ is the number of ways to choose T_i with labels $\underline{n}$.

Proof: You should be able to prove the Rule of Sum. The Rule of Product requires more work. Let t_n be the number of $T \in \mathcal{T}$ with $w(T) = n$. By the assumptions,

$$t_n \;=\; \sum_{K_1 \subseteq \underline{n}} t_{1,|K_1|}\; t_{2,n-|K_1|}$$

and so

$$\frac{t_n}{n!} \;=\; \frac{1}{n!}\sum_{k=0}^{n} \binom{n}{k} t_{1,k}\; t_{2,n-k} \;=\; \sum_{k=0}^{n} \frac{t_{1,k}}{k!}\,\frac{t_{2,n-k}}{(n-k)!},$$

a convolution. Multiply by x^n, sum over n and rearrange to obtain $\mathsf{E}_{\mathcal{T}}(x) = \mathsf{E}_1(x)\mathsf{E}_2(x)$. □

If you compare these theorems with those given for ordinary generating functions in Section 10.4, you may notice some differences.

- First, the Rule of Product was stated here for a sequence of only two choices. This is not an essential difference—you can remove the constraint at the expense of a more cumbersome statement of the theorem or you can simply iterate the theorem: divide things into two choices, then subdivide the second choice in two and so on.
- The second difference seems more substantial: There is no parallel to condition (iii) of the ordinary generating function version, Theorem 10.4 (p. 292). This is because it is implicitly built into the statement of the theorem—the EGF counts by total number of labels, so it follows that if T comes from T_1 and T_2, then $w(T_1) + w(T_2) = w(T)$.
- Finally, neither of the theorems here mentions either an infinite number of blocks in the partition (Rule of Sum) or an infinite number of steps (Rule of Product). Again, this is not a problem—infinitely many is allowed here, too.

Example 11.4 Counting derangements Recall that a derangement is a permutation with no fixed points and that D_n denotes the number of derangements on an n element set. It is convenient to say that there is one permutation of the empty set and that it is a derangement because it does not map anything to itself. By splitting off the fixed points from an arbitrary permutation, say $n-k$ of them, we have

$$n! \;=\; \sum_{k=0}^{n} \binom{n}{k} D_k. \tag{11.16}$$

We can manipulate this to obtain an EGF for D_n, but it is easier to go directly from the combinatorial argument to the Rule of Product. We'll do that now.

Let $D(x)$ be the EGF for derangements by number of things deranged. A permutation of $\underline{n}$ can be constructed by choosing some subset K of $\underline{n}$, constructing a derangement of K and fixing the elements of $\underline{n} - K$. Every permutation of $\underline{n}$ arises exactly once this way. We make three simple observations:

- The EGF for all permutations is
$$\sum_{n=0}^{\infty} n!(x^n/n!) \;=\; \frac{1}{1-x}.$$
- Since there is just one permutation fixing all the elements of a given set, the EGF for permutations with all elements fixed is $\sum x^n/n! = e^x$.
- By the Rule of Product, $1/(1-x) = D(x)e^x$ and so
$$D(x) \;=\; \frac{e^{-x}}{1-x}. \tag{11.17}$$

Note that we never needed to write down (11.16). In Example 10.7 (p. 283) and Exercise 10.3.1 (p. 291), (11.17) was used to obtain simple recursions for D_n.

The simplicity of this derivation and the ones in the examples that follow illustrate the power of the Rule of Product. □

Example 11.5 Sequences of letters How many n long sequences can be formed from A, B and C so that the number of A's in the sequence is odd and the number of B's in the sequence is odd? The labeled objects are simply the positions in the sequence. We form an ordered partition of the labels $\underline{n}$ into three parts, say (P_A, P_B, P_C). The letter A is placed in all the positions that are contained in the set P_A, and similarly for B and C. This is just the set up for the Rule of Product. If $|P_A|$ is odd, we can place the A's in just one way, while if $|P_A|$ is even, we cannot place them because of the requirement that the sequence contain an odd number of A's. Thus the EGF for the A's is

$$\sum_{k \text{ odd}} x^k/k! \;=\; (e^x + e^{-x})/2.$$

This is also the EGF for the B's. For the C's we have $\sum x^k/k! = e^x$. Thus the EGF for our sequences is

$$\left(\frac{e^x + e^{-x}}{2}\right)^2 e^x \;=\; \frac{e^{3x} + 2e^x + e^{-x}}{4},$$

and so the answer is $(3^n + 2 + (-1)^n)/4$. You might like to try to find a direct counting argument for this result.

This could also have been done with ordinary generating functions and multisection of series: Keep track of the number of A's, the number of B's and the length of the sequence using three variables in the generating function. Then use multisection twice to insure an odd number of A's and an odd number of B's. Finally, set the variables that are keeping track of the A's and B's equal to 1. We will not carry out the details. □

As noted in the last example, some problems can be done with both ordinary and exponential generating functions. In such cases, it is usually clear that one method is easier than the other. In some other problems, it is necessary to use generating functions that are simultaneously exponential and ordinary. This happens because one class of objects we're keeping track of has labels and the other class does not. Here's an example of this.

Example 11.6 Words from a collection of letters In Example 1.11 (p. 13) we considered the problem of counting strings of letters of length k, where the letters can be repeated but the number of repetitions is limited. Specifically, we used the letters in the word ERROR but insisted no letter could be used more than it appeared. We suggest that you review Example 1.11 (p. 13) and the improved methods in Examples 1.19 (p. 24) and 3.3 (p. 69), where we used the letters in ERRONEOUSNESS. Imagine letters that are labeled, each by its position in the word. Since there are three E's and only one word of any length can be built with just E's, the EGF for words of E's is $1 + x + x^2/2 + x^3/6$. Choose E's and R's and O's and N's and U's and S's. In this the generating function for words is

$$\left(1 + x + \frac{x^2}{2} + \frac{x^3}{6}\right)^2 \left(1 + x + \frac{x^2}{2}\right)^3 (1 + x)$$
$$= 1 + 6x + \frac{35x^2}{2} + \frac{197x^3}{6} + \frac{265x^4}{6} + 45x^5 + \frac{322x^6}{9} + \frac{811x^7}{36}$$
$$+ \frac{541x^8}{48} + \frac{40x^9}{9} + \frac{389x^{10}}{288} + \frac{29x^{11}}{96} + \frac{13x^{12}}{288} + \frac{x^{13}}{288},$$

where the multiplication was done by a symbolic manipulation package. Multiplying the coefficient of x^n by $n!$ gives the number of n-long words. Hence the number of 8-long words is 454,440. □

Example 11.7 Set partitions We want to count the number of partitions of a set, keeping track of the size of the set that is being partitioned and the number of blocks in the partition. Since the elements of the set are distinct, they are labeled and so we will use an EGF to count them. On the other hand, the blocks of a partition are not labeled, so it is natural to use an ordinary generating function to count blocks.

Let's start by looking at partitions with one block. There is just one such, so the EGF is $\sum_{n>0} x^n/n! = e^x - 1$.

What about partitions with two blocks? We can use the Rule of Product. In fact, the statement of the Rule of Product has built into it partitions of the set into two blocks K and L. Thus the EGF should be $(e^x - 1)^2$. This is not quite right because these blocks are *ordered* but the blocks of a partition of a set are supposed to be unordered. As a result, we must divide by 2!.

You should have no trouble showing that the number of partitions of a set that have exactly k blocks has the EGF $(e^x - 1)^k/k!$.

Recall that $S(n, k)$, the Stirling number of the second kind, is the number of partitions of an n element set into exactly k blocks. By the previous paragraph,

$$\sum_n S(n, k)x^n/n! \;=\; \frac{(e^x - 1)^k}{k!}. \tag{11.18}$$

Let $S(0, 0) = 1$. It follows that

$$\sum_{n,k} S(n, k)(x^n/n!)y^k \;=\; \sum_{k=1}^{\infty} \frac{(e^x - 1)^k}{k!} y^k + S(0, 0) \;=\; \exp(y(e^x - 1)).$$

($\exp(z)$ is another notation for e^z.) Call this $A(x, y)$.

The formula for $A(x, y)$ can be manipulated in various ways to obtain recursions, formulas and other relations for $S(n, k)$. Of particular interest is the total number of partitions of a set, which is

given by $B_n = \sum_k S(n,k)$. You should be able to see that we can obtain its generating function by setting $y = 1$. Thus

$$\begin{aligned}\sum_n B_n x^n/n! &= A(x,1) = \exp(e^x - 1) \\ &= e^{-1}\exp(e^x) = \frac{1}{e}\sum_{i=0}^{\infty}\frac{(e^x)^i}{i!} = \frac{1}{e}\sum_{i=0}^{\infty}\frac{e^{ix}}{i!} \\ &= \frac{1}{e}\sum_{i=0}^{\infty}\frac{1}{i!}\sum_{n=0}^{\infty}\frac{(ix)^n}{n!} = \sum_{n=0}^{\infty}\left(\frac{1}{e}\sum_{i=0}^{\infty}\frac{i^n}{i!}\right)\frac{x^n}{n!}.\end{aligned}$$

Thus we obtain

Theorem 11.3 Dobinski's formula *The number of partitions of $\underline{n}$ is*

$$\frac{1}{e}\sum_{i=0}^{\infty}\frac{i^n}{i!}. \quad \square$$

Example 11.8 Mixed generating functions Suppose we are keeping track of both labels and unlabeled things. For example, we might count set partitions keeping track of both the size of the set and the number of blocks. We state without proof

Theorem 11.4 Mixed Generating Functions *If we are keeping track of more that one thing, some of which are labeled and some of which are unlabeled, then the Rules of Sum and Product still apply. For the Rule of Product, the labels must satisfy the conditions in Theorem 11.2 and the remaining weights must satisfy the conditions in Theorem 10.4 (p. 292).*

Returning to the set partition situation, if y keeps track of the number of blocks, then (11.18) becomes

$$y^k \sum_n S(n,k)x^n/n! = \frac{y^(e^x - 1)^k}{k!}$$

and so the generating function is $\sum S(n,k)x^n y^k/n! = \exp(y(e^x - 1))$. $\square$

The Exponential Formula

Before stating the Exponential Formula, we'll look at a special case.

Example 11.9 Connected graphs Let g_n be the number of simple graphs with vertex set $\underline{n}$ and let c_n the number of such graphs which are connected. Recall that a simple graph is a graph whose edge set is a subset of $\mathcal{P}_2(V)$. Since each pair of vertices is either an edge or not and there are $\binom{n}{2}$ pairs of vertices, it follows that $g_n = 2^{\binom{n}{2}}$. (This is *not* $2\binom{n}{2}$; the $\binom{n}{2}$ is an exponent.) What is the value of c_n?

This problem can be approached in various ways. We'll look at it in a way that can be generalized considerably so that we'll be able to obtain other results. The basic idea is to view a graph as being built from connected components. We must either

- figure out how a graph decomposition into connected components translates into a generating function or
- find some way to distinguish a component so we can split off one component and thus proceed in a recursive fashion.

We'll follow the latter approach. In order to distinguish a component, we will root the graph.

Let $G(x)$ and $C(x)$ be the EGFs for g_n and c_n. It will be convenient to take $g_0 = 1$ and $c_0 = 0$.

Imagine rooting the graph by choosing some vertex to be the root. There are n distinct ways to do this and so there are ng_n such graphs. Thus the generating function for the rooted graphs is $xG'(x)$.

We can construct a rooted graph by choosing a rooted component and then choosing the rest of the graph. This is just the set up for the Rule of Product. Rooted components have the EGF $xC'(x)$. The rest of the rooted graph is simply a graph and so its generating function is $G(x)$. (Note that this works even when the rest of the rooted graph is empty because we have $g_0 = 1$.) We have proved that

$$xG'(x) \;=\; \big(xC'(x)\big)\, G(x). \tag{11.19}$$

Ignoring questions of convergence, as we usually do, we can easily solve this differential equation by separation of variables to obtain

$$C(x) \;=\; \ln\big(G(x)\big) + A,$$

where the constant A needs to be determined. (Here's how separation of variables works in this case. We have $xdG/dx = (xdC/dx)G$ and so $dG/G = dC$, which we integrate.)

Since $C(0) = c_0 = 0$ and $G(0) = g_0 = 1$, it follows that $A = 0$. Thus we have $G(x) = \exp(C(x))$. It would be nice if this formula led to a simple method for calculating $C(x)$. This is not the case—in fact, it is easier to equate coefficients of x^n in (11.19) and rearrange it into a (rather messy) recursion for c_n. □

The formula $G = e^C$ has been generalized considerably in the research literature. Here is one form of it.

Theorem 11.5 Exponential Formula *Suppose that we have two sets $\mathcal{S}$ and $\mathcal{C}$ of labeled structures. A structure is rooted by distinguishing one its labels and calling it the root. Form all possible rooted structures from those in $\mathcal{S}$ and call the set $\mathcal{S}_r$. Do the same for $\mathcal{C}$. If the Rule of Product holds so that $\mathsf{E}_{\mathcal{S}_r} = \mathsf{E}_{\mathcal{C}_r}\mathsf{E}_{\mathcal{S}}$, then*

$$\mathsf{E}_{\mathcal{S}} \;=\; \exp\big(\mathsf{E}_{\mathcal{C}}\big).$$

($\exp(z)$ is another notation for e^z.) The proof is the same as that for $G = e^C$.

Example 11.10 Graphs revisited The previous example can be extended in two important directions.

More parameters: We could include more variables in our generating functions to keep track of other objects besides number of vertices. The basic requirement is that the number of such objects in our rooted graph must equal the number in its rooted component plus the number in the rest of the graph so that we still have the differential equation $x\frac{\partial G}{\partial x} = (x\frac{\partial C}{\partial x})G$. (Of course, G and C now contain other variables besides x and so the derivative with respect to x is a partial derivative.) You should be able to easily see that we still have the solution $G = \exp(C)$, either from the differential equation or from Theorem 11.5. Let's look at a couple of examples.

- We can keep track of the number of components in a graph. Let $g_{n,k}$ be the number of simple graphs with $V = \underline{n}$ that have exactly k components. The generating function is $G(x,y) = \sum_{n,k} g_{n,k}(x^n/n!)y^k$ because the vertices are labeled but the components are not. Of course, a connected graph has exactly one component. Thus $C(x,y) = C(x)y$. We have $G(x,y) = \exp(C(x)y)$. Since $\exp(C(x)) = G(x)$, it follows that

$$G(x,y) \;=\; G(x)^y. \tag{11.20}$$

  What does this expression mean; that is, how should one interpret $G(x)^y$?

  We can write $G(x) = 1 + xH(x)$ for some power series $H(x)$. By the binomial theorem for arbitrary powers we have

$$G(x)^y \;=\; \sum_{k=0}^{\infty} \binom{y}{k} x^k H(x)^k. \tag{11.21}$$

  This expression makes perfectly good sense: We have

$$\binom{y}{k} \;=\; \frac{y(y-1)\cdots(y-k+1)}{k!},$$

  and if we want a coefficient, say that of $x^n y^m$, we need only look at a finite number of terms, namely those with $k \le n$. (You may be concerned that (11.21) is really the same as $G(x,y) = \exp(yC(x))$. It is. This can be shown by formal power series manipulations.)

  It may appear that (11.20) gives us something for nothing—just by knowing the number of graphs by vertices we can determine the number of graphs by both vertices and components. Unfortunately, we don't get this for nothing because calculation of numerical values for (11.20) can involve a fair bit of work.

- We can keep track of the number of edges in a graph. Let $g_{n,q}$ be the number of simple graphs with $V = \underline{n}$ that have exactly q edges and define $c_{n,q}$ similarly. Since each of the $\binom{n}{2}$ elements of $\mathcal{P}_2(V)$ may be an edge or not, $\sum_q g_{n,q} z^q = (1+z)^{\binom{n}{2}}$. Thus

$$G(x,z) \;=\; \sum_{n\ge 0}(x^n/n!)\sum_{q\ge 0} g_{n,q} z^q \;=\; \sum_{n=0}^{\infty}(1+z)^{\binom{n}{2}}(x^n/n!)$$

  and $C(x,z) = \ln(G(x,z))$. Unfortunately, no simple formula is known for the sum.

Special collections of graphs: We can limit our attention to particular subsets of the set of all labeled graphs. How is this done?

Let $\mathcal{C}$ be some set of connected graphs and let $G(\mathcal{C})$ be all the graphs whose connected components lie in this collection. Suppose the collection $\mathcal{C}$ satisfies the condition that the number of n vertex graphs in the set $\mathcal{C}$ depends on n and not on the labels of the vertices. In this situation, we can still derive our equation $G = e^C$ and can still keep track of other objects besides vertices. Let's look at some examples.

- Suppose that the connected components are complete graphs; i.e., every vertex in the component is connected to every other. The only thing that distinguishes one component from another is

its set of vertices. Thus we can identify such a graph with a partition of its vertex set in which each block corresponds to the vertex set of a component. This is a bijection between this class of graphs and partitions of sets. We easily have $c_n = 1$ for all $n > 0$ and so $C(x) = e^x - 1$. Consequently, $G(x) = \exp(e^x - 1)$. The number of components of such a graph corresponds to the number of blocks in the partition. Thus $G(x, y) = \exp((e^x - 1)y)$, in the notation of (a), is the generating function for the Stirling numbers of the second kind. Hence Example 11.7 is a special case of our $G = e^C$ formula.

- Our next illustration, cycles of a permutation, will be a separate example. □

Example 11.11 Permutations and their cycles Let L be a set of positive integers and let s_n be the number of permutations of $\underline{n}$ such that all the cycle lengths are in L. Some examples are

(a) L is all positive integers so s_n counts all permutations;

(b) $L = \{2, 3, \ldots\}$ so s_n counts derangements;

(c) $L = \{1, 2\}$ so s_n counts involutions.

We'll obtain a generating function for s_n. In this case, the labels are simply the integers $\underline{n}$ that are being permuted.

One approach is to draw a directed graph with vertex set $\underline{n}$ and an edge (i, j) if the permutation maps i to j. This is a graph whose components are cycles and the lengths of the cycles are all in L. We can then use the approach in the previous example. This is left for you to do. We'll "forget" the previous example and go directly to the Exponential Formula.

We can construct a permutation, by choosing its cycles. Let a structure in $\mathcal{C}$ be a cycle. The parts of the structure are the things permuted by the cycle. Let a structure in $\mathcal{S}$ be a permutation. When a permutation is rooted by choosing an element of $\underline{n}$, it breaks up into a rooted cycle and an unrooted permutation. Thus the Exponential Formula can be used.

Let c_n be the number of n long cycles that can be made from $\underline{n}$. By the Exponential Formula,

$$C(x) \;=\; \mathsf{E}_{\mathcal{C}}(x) \;=\; \sum_{n\geq 1} c_n \,\frac{x^n}{n!}$$

and

$$S(x) \;=\; \mathsf{E}_{\mathcal{S}}(x) \;=\; e^{C(x)} \;=\; \exp\Bigl(\sum_{n\geq 1} c_n \,\frac{x^n}{n!}\Bigr).$$

We need to determine c_n. Since all cycle lengths must lie in L, $c_n = 0$ when $n \notin L$. Suppose $n \in L$. To construct a cycle $f\colon \underline{n} \to \underline{n}$, we specify $f(1)$, $f^2(1) = f(f(1))$, $f^3(1)$, …, $f^{n-1}(1)$. Since we want a cycle of length n, $f^n(1) = 1$ and the values $1, f(1), f^2(1), \ldots, f^{n-1}(1)$ are all distinct. Since these are the only conditions, $f(1), f^2(1), \ldots, f^{n-1}(1)$ can be any permutation of $2, 3, \ldots, n$. Thus $c_n = (n-1)!$. It follows that

$$S(x) \;=\; \exp\Bigl(\sum_{k\in L} x^k/k\Bigr). \tag{11.22}$$

Let's reexamine the three examples of L that were mentioned above. Let y stand for the sum in (11.22).

(a) When L is all positive integers,

$$y \;=\; \sum_{k=1}^{\infty} \;=\; -\ln(1-x)$$

and so $S(x) = 1/(1-x) = \sum_{n\geq 0} x^n = \sum_{n\geq 0} n!x^n/n!$. Hence $s_n = n!$, which we already knew—there are $n!$ permutations of $\underline{n}$.

(b) For derangements, y equals its value in (a) minus x and so

$$S(x) \;=\; \frac{e^{-x}}{1-x},$$

which we obtained in Example 11.4 (p. 318) by other methods.

(c) For involutions, $y = x + x^2/2$ and so

$$S(x) \;=\; \sum_{k=0}^{\infty} \frac{(x+x^2/2)^k}{k!} \;=\; \sum_{k=0}^{\infty}\sum_{j=0}^{k} \binom{k}{j} \frac{x^{k-j}(x^2/2)^j}{k!}.$$

Collecting the terms with $k + j = n$, we obtain

$$s_n \;=\; \sum_{j=0}^{n} \frac{n!}{j!\,2^j(n-2j)!},$$

which we obtained by a counting argument in Theorem 2.2 (p. 48). □

Example 11.12 Permutations and their cycles (continued) We can keep track of other information as well, provided it suffices to look at each cycle separately. In an extreme case, we could keep track of the number of cycles of each length. This requires an infinite number of classes of unlabeled parts, one for each cycle length. Let the associated variable for cycles of length k be z_k. The resulting generating function will be an EGF in the size of the set being permuted, but an ordinary generating function in each z_k since cycles are not labeled. You should be able to show that the generating function is

$$\exp\left(\sum_{k\in L} \frac{z_k x^k}{k}\right). \tag{11.23}$$

If we simply keep track of the size of the set being permuted and the number of cycles in the permutation, the generating function is just the $G(x, y)$ of (11.20). You should be able to see why this is so . If not, the second paragraph of the previous example may help. Another way to see it is to use (11.23) with $z_k = y$ for all k. Let $z(n, k)$ be the number of permutations of $\underline{n}$ that have exactly k cycles. These are called the *signless Stirling numbers of the first kind*. We have just seen that

$$Z(x,y) \;=\; \sum_{n,k\geq 0} z(n,k)(x^n/n!)y^k \;=\; \left(\frac{1}{1-x}\right)^y \;=\; (1-x)^{-y}. \tag{11.24}$$

Equating coefficients of $x^n/n!$, we obtain

$$\sum_k z(n,k)y^k \;=\; (-1)^n n! \binom{-y}{n} \;=\; y(y+1)\cdots(y+n-1).$$

We can use our generating function to study the expected number of cycles. The method for doing so was worked out in (10.21) (p. 284). Review that equation. Now the random variable X_n is the number of cycles and our generating function is $Z(x, y)$ instead of $A(x, y)$ and it is exponential in x. Since it is exponential, we should have a factor of $n!$ in both the numerator and denominator of (10.21) but, since it appears in both, it cancels out. On with the calculations! $Z(x, 1) = (1-x)^{-1} = \sum_{n\geq 0} x^n$, so the denominator in (10.21) is 1. We have $Z_y(x, y) \;=\; \ln(1-x)(1-x)^{-y}$ and so

$$Z_y(x,1) \;=\; \frac{1}{1-x}(-\ln(1-x)) \;=\; \frac{1}{1-x}\sum_{k\geq 1}\frac{x^k}{k}$$

by Taylor's theorem for $-\ln(1-x)$. Finally $[x^n]\,Z_y(x,1) = \sum_{k=1}^{n} \frac{1}{k}$, which you can work out by expanding (11.24) further or by using Exercise 10.1.6 (p. 274).

We've just worked out that the average number of cycles in a permutation of $\underline{n}$ is $\sum_{k=1}^{n} \frac{1}{k}$, a result that was derived by other means in Example 2.8 (p. 47). Using Riemann sum approximations as we did in deriving (10.30) from (10.29), it follows that the average number of cycles in a permutation of $\underline{n}$ is $\ln n + O(1)$ as $n \to \infty$.

Let's work out the variance using (10.22) (p. 285). We have $Z_{yy}(x,y) = (-\ln(1-x))^2(1-x)^{-y}$. Proceeding as in the previous paragraph,

$$[x^n]\, Z_{yy}(x,1) \;=\; \sum_{k=0}^{n} [x^k]\,(-\ln(1-x))^2 \;=\; \sum_{k=0^n} \sum_{i+j=k} \frac{1}{i}\,\frac{1}{j} \;=\; \sum_{i+j\le n} \frac{1}{i}\,\frac{1}{j}.$$

Thus

$$\begin{aligned} \mathbf{var}(X_n) \;&=\; \sum_{i+j\le n} \frac{1}{i}\,\frac{1}{j} + \mathbf{E}(X_n) - \left(\sum_{i=1}^{n} \frac{1}{i}\right)^2 \\ &=\; \sum_{i+j\le n} \frac{1}{i}\,\frac{1}{j} + \mathbf{E}(X_n) - \sum_{i,j\le n} \frac{1}{i}\,\frac{1}{j} \;=\; \mathbf{E}(X_n) - \sum_{\substack{i,j\le n \\ i+j>n}} \frac{1}{i}\,\frac{1}{j}. \end{aligned}$$

We've already done a lot of computations, so we'll just remark without proof that

$$\sum_{\substack{i,j\le n \\ i+j>n}} \frac{1}{i}\,\frac{1}{j} \;\sim\; \int_0^1 \frac{-\ln(1-x)}{x}\,dx \;=\; \sum_{n=1}^{\infty} \frac{1}{n^2} \;=\; \frac{\pi^2}{6}.$$

Thus $\mathbf{var}(X_n) \sim \mathbf{E}(X_n) \sim \ln n$. By Chebyshev's inequality (C.3) (p. 385), it's unlikely that $|X_n - \ln n|/(\ln n)^{1/2}$ will be large. □

*Example 11.13 Permutations and their cycles (concluded) You should review the discussion of parity of a permutation in Definition 2.4 (p. 49) and Theorem 2.3. Let L be a set of positive integers. Let e_n (resp. o_n) be the number of even (resp. odd) permutations of $\underline{n}$ all of whose cycle lengths are in L. Let $p_n = e_n - o_n$. Using the previous example and Theorem 2.3(c), you should be able to show that the exponential generating function for p_n is given by

$$P(x) \;=\; \exp\!\left(\sum_{k\in L} \frac{(-1)^{k-1}x^k}{k}\right).$$

Let's look at some special cases.

If L be the set of all positive integers, then

$$\sum_{k\in L} \frac{(-1)^{k-1}x^k}{k} \;=\; \sum_{k=1}^{\infty} \frac{(-1)^{k-1}x^k}{k} \;=\; \ln(1+x)$$

and so $P(x) = 1 + x$. In other words, when $n > 1$, there are as many even permutations of $\underline{n}$ as there are odd and so $e_n = o_n = n!/2$. We already knew this in Theorem 2.3.

If $L = \{2, 3, \ldots\}$, we are looking at derangements. In this case

$$\sum_{k\in L} \frac{(-1)^{k-1}x^k}{k} \;=\; \sum_{k=1}^{\infty} \frac{(-1)^{k-1}x^k}{k} - x \;=\; \ln(1+x) - x$$

and so $P(x) = (1+x)e^{-x}$. With a little algebra, $p_n = (-1)^{n-1}(n-1)$. Thus the number of even and odd derangements of $\underline{n}$ differ by $n-1$, and there are more even derangements if and only if n is odd. This is a new result for us, and it's not clear how to go about proving it without generating functions.

Suppose L consists of all positive integers except $k > 2$. Reasoning as in the previous paragraph, $P(x) = (1+x)e^{\pm x^k/k}$, where the sign is plus when k is odd. If you expand $P(x)$ in a power series, you should see that $pn \neq 0$ if and only if n is a multiple of k or one more than a multiple of k. We

have shown that, for $k > 2$, among the permutations of $\underline{n}$ with no k-cycles, the numbers of even and odd permutations are equal if and only if neither n nor $n-1$ is a multiple of k. □

Example 11.14 Rooted labeled trees Let's study t_n, the number of labeled rooted trees with $V = \underline{n}$.

If we remove the root vertex from a rooted tree, making all of its sons new root vertices, we obtain a graph all of whose components are rooted labeled trees. This process is reversible.

Let T be the EGF for the t_n's. By the Exponential Formula, the generating function for graphs all of whose components are rooted labeled trees is e^T. Call all such graphs $\mathcal{F}$. Thus $\mathsf{E}_{\mathcal{F}} = e^T$. The process we described in the previous paragraph constructs a rooted tree by

- partitioning the labels into (K, L) with $|K| = 1$,
- assigning the label in K to the new root,
- choosing an element $F \in \mathcal{F}$ with labels L, and
- joining the roots of the trees in F to the new root.

By the Rule of Product $T = x\mathsf{E}_{\mathcal{F}} = xe^T$.

How can we obtain t_n from the equation $T = xe^T$? There is a technique, called *Lagrange inversion*, which can be useful in this situation.

Theorem 11.6 Lagrange Inversion *Let $T(y)$, $f(y)$ and $g(y)$ be power series with $f(0) \neq 0$. Suppose that $T(x) = xf(T(x))$. Then the coefficient of x^n in $g(T(x))$ is the coefficient of u^{n-1} in $g'(u)(f(u))^n/n$; that is, $[x^n]\, g(T(x)) = [u^{n-1}]\left(g'(u)f(u)^n/n\right)$.*

Proofs and generalizations of Lagrange inversion are discussed at the end of this chapter.

In our particular case, $g(u) = u$ and $f(u) = e^u$. Thus

$$t_n/n! = [u^n]\, e^{nu}/n = (n^n/n!)/n.$$

Thus $t_n = n^{n-1}$.

Incidentally, we choose the symbol $\mathcal{F}$ because graphs whose components are trees are called *forests*. □

Exercises

*11.2.1. Let $\mathcal{T}$ be a collection of structures, each of which can have labeled unlabeled parts. An example is the partitions of an n-set counted by $n > 0$ and by the number of blocks. Let $T(x, y)$ be the generating function for $\mathcal{T}$, where it is exponential in the labeled parts (using x) and ordinary in the unlabeled parts (using y). State and prove a Rule of Product for these generating functions.

11.2.2. Let $\mathcal{T}$ be a collection of structures, each of which has at least one labeled part. Let $\mathsf{E}_{\mathcal{T}}$ be the EGF for $\mathcal{T}$, with respect to these labeled parts. Prove the following results and compare them with those in Exercise 10.4.1 (p. 298). When talking about lists (or sets) of structures in this exercise, the labels that are used for the objects all differ; that is, structures at different positions in a list (or set) must have different labels. If a totality of n objects in a class appear, they are to use the labels in $\underline{n}$.

(a) The EGF for k-lists of structures is $\left(\mathsf{E}_{\mathcal{T}}\right)^k$.

(b) The EGF for lists of structures is $(1 - \mathsf{E}_{\mathcal{T}})^{-1}$. Here lists of any length are allowed, including the empty list.

(c) The EGF for sets of structures, where each structure must come from $\mathcal{T}$ is $\exp\left(\mathsf{E}_{\mathcal{T}}\right)$.

(d) The EGF for circular lists is $-\ln\left(1 - \mathsf{E}_{\mathcal{T}}\right)$.

These results could be generalized to allow for unlabeled parts by using the result of Exercise 11.2.1.

11.2.3. Let $z(n,k)$ be the signless Stirling numbers of the first kind, defined in Example 11.12.

(a) Let $Z_n(y) = \sum_k z(n,k)y^k$. Show that $Z_n(y) = Z_{n-1}(y) \times (y+n-1)$ and use this to deduce the recursion

$$z(n,k) \;=\; z(n-1,k-1) + (n-1)z(n-1,k).$$

(b) Prove the previous recursion by a direct counting argument using the fact that $z(n,k)$ is the number of permutations of $\underline{n}$ with exactly k cycles.

11.2.4. Let a_n be the number of ways to place n labeled balls into boxes numbered $1, 2, \cdots$ so that the number of balls in the kth box is a multiple of k. (This is the labeled analog of the box–ball interpretation of partitions of a number.) The EGF for a can be written as a product of sums. Do it.

11.2.5. Let a_n be the number of sequences of A's B's and C's such that any letter that actually appears in the sequence must appear an odd number of times.

(a) Show that the EGF for a_n is $\left(1 + \dfrac{e^x - e^{-x}}{2}\right)^3$.

(b) Show that for n odd $a_n = (3^n + 9)/4$ and for $n > 0$ and even $a_n = 3 \times 2^{n-1}$.

11.2.6. Let $a_{n,k}$ be as in Exercise 11.2.5, except that k different letters are used instead of just A, B and C.

(a) Obtain a generating function $\sum_n a_{n,k}x^n/n!$.

(b) Show that $a_{n,k}$ equals $\sum \binom{k}{j} 2^{-j} c_{n,j}$, where the sum is over all j with the same parity as n and $\sum_n c_{n,j}x^n/n! = (e^x - e^{-x})^j$.

*(c) Can you obtain a more explicit formula for $a_{n,k}$?

11.2.7. Recall that B_n is the number of partitions of the set $\underline{n}$. Let $B(x)$ be the EGF. Show that $B'(x) = e^x B(x)$ and use this to show that

$$B_{n+1} \;=\; \sum_{k=0}^{n} \binom{n}{k} B_k.$$

11.2.8. Let a_n be the number of partitions of $\underline{n}$ such that each block has an odd number of elements and let $A(x)$ be the EGF. Use the Exponential Formula to show that $A(x) = \exp(e^x - e^{-x})/2)$.

11.2.9. Let $C(x)$ and $G(x) = \exp(C(x))$ be the EGFs for some collection of connected graphs and some collection of graphs, respectively.

(a) Let $H(x) = C(x)G(x)$ be an EGF for the sequence h_n. Using the Exponential Formula, show that the average number of components in the graphs counted by g_n is h_n/g_n.

(b) Prove the formula in the previous part by a simple application of the Rule of Product.
Hint. Remember that $H(x)$ will be counting the *total* number of components, not the average number.

(c) Deduce the formula at the end of Example 11.12 for the average number of cycles in a permutation.

(d) Deduce that the average number of blocks in a partition of $\underline{n}$ is $\frac{B_{n+1}}{B_n} - 1$.

(e) Obtain a formula like the previous one for the average number of cycles in an involution.

11.2.10. Let a_n be the number of partitions of $\underline{n}$ where the order of the blocks is important. Obtain the EGF $A(x)$ and use it to show that $a_n = \sum_{k>0} k^n/2^{k+1}$.

11.2.11. A permutation f of $\underline{n}$ is said to be *alternating* if $f(1) < f(2) > f(3) < \ldots$. Let a_n be the number of alternating permutations of $\underline{n}$. It will be convenient to set $a_0 = 1$. Let A be the EGF for the a_n's and let B be the EGF for those a_n with odd n; that is, $b_{2n} = 0$ and $b_{2n+1} = a_{2n+1}$. By considering $f(1), \ldots, f(k-1)$ and $f(k+1), \ldots, f(n)$, where $k = f^{-1}(n)$, show that

$$B'(x) \;=\; B(x)^2 + 1 \;\text{ and }\; A'(x) = B(x)A(x) + 1.$$

Verify that $A(x) = \tan x + \sec x$ and $B(x) = \tan x$.

11.2.12. A k-ary tree is a rooted tree in which each nonleaf vertex has exactly k sons. Let t_n be the number of unlabeled plane k-ary trees having n nonleaf vertices.

(a) Prove that the ordinary generating function for t_n satisfies the equation $T(x) = 1 + xT(x)^k$.

(b) Use Lagrange inversion (Theorem 11.6) to show that $t_n = \frac{1}{n}\binom{nk}{n-1}$.
Hint. Apply the theorem to $S(x) = T(x) - 1$.

11.2.13. A function $f: A \to A$ is said to have a square root if there is a function $g: A \to A$ such that $g(g(a)) = f(a)$ for all $a \in A$.

(a) Show that a permutation has a square root if and only if, it has an even number of cycles of length $2k$ for each $k > 0$.

(b) Let s_n be the number of permutations of $\underline{n}$ which have a square root. Show that

$$S(x) = \sum_{n\geq 0} s_n x^n/n! = \exp\Bigg(\sum_{\substack{k=1\\ k \text{ odd}}}^{\infty} \frac{x^k}{k}\Bigg) \prod_{\substack{k=2\\ k \text{ even}}}^{\infty} \frac{\exp(x^k/k) + \exp(-x^k/k)}{2}.$$

Hint. One way is to use $\exp\Big(\sum_{k=1}^{\infty} z_k x^k/k\Big)$ and then use bisection of series for each even k one by one. Another approach is to do each cycle length separately and then use the Rule of Product.

(c) Using $\cosh(u) = (e^u + e^{-u})/2$ and bisection of the Taylor series for $\ln(1-x)$, show that

$$S(x) = \sqrt{\frac{1+x}{1-x}} \prod \cosh(x^{2k}/2k).$$

(d) By taking logarithms, differentiating and then multiplying by $S(x)$ conclude that $S'(x) = S(x)B(x)$ where $B(x)$ is $(1-x^2)^{-1}$ plus the sum of $x^{n-1}\tanh(x^n/n)$ over all even positive n. How much work is involved in using this to construct tables of s_n? Can you think of an easier method?

11.2.14. Let $a_{n,k}$ be the number of permutations of $\underline{n}$ having exactly k cycles of even length.

(a) Show that

$$\sum_{n,k} \frac{a_{n,k} x^n y^k}{n!} = (1-x^2)^{-y/2} \sqrt{\frac{1+x}{1-x}}.$$

(b) Conclude that the average number of even length cycles in a permutation of $\underline{n}$ is

$$\sum_{k=1}^{\lfloor n/2 \rfloor} \frac{1}{2k},$$

where the floor function $\lfloor x \rfloor$ is the largest integer not exceeding x.

(c) Show that the number of odd length cycles minus the number of even length cycles averaged over all permutations of $\underline{n}$ is

$$\sum_{k=\lfloor n/2 \rfloor + 1}^{n} \frac{1}{k}$$

and that this sum approaches $\ln 2$ as $n \to \infty$.

*11.2.15. Let $t_{n,k}$ be the number of n-vertex rooted labeled trees with k leaves. Let $L(x,y) = \sum_{n,k} t_{n,k}(x^n/n!)y^k$. (The generating function for leaves is ordinary because the labels on the leaves have already been taken into account when we counted vertices.) Let $T(x)$ be the EGF for rooted labeled trees by number of vertices.

(a) Show that $L = xe^L - x + xy$.

(b) Let $U(x)$ be the EGF $\left.\frac{\partial L(x,y)}{\partial y}\right|_{y=1}$. Show that the average number of leaves in an n-vertex rooted labeled tree is u_n/n^{n-1}.

*(c) Show that $U(x) = x^2T'(x) + x$ and use this to show that the average number of leaves in an n-vertex rooted labeled tree is $n(1-1/n)^{n-1}$. Conclude that the probability that a randomly chosen vertex is a leaf approaches $1/e$ as $n \to \infty$.

*11.2.16. In this exercise, you'll study the average height of vertices in rooted labeled trees. The height of a vertex is the number of edges on the path from it to the root. For a rooted tree T, let $h(T)$ be the sum of the heights of the vertices of T. Let $t_{n,k}$ be the number of n-vertex rooted labeled trees T with $h(T) = k$ and let $H(x,y) = \sum_{n,k} t_{n,k}(x^n/n!)y^k$. If T has n vertices, the average height of a vertex in T is $h(T)/n$. Let $\mu(n)$ be the average of $h(T)/n$ over all n-vertex rooted labeled trees.

(a) Show that

$$\mu(n) = n^{-n}\sum_k kt_{n,k}$$

and that $n^n\mu(n)$ is the coefficient of $x^n/n!$ in $D(x) = \left.\frac{\partial H(x,y)}{\partial y}\right|_{y=1}$.

(b) Show that $H(x,y) = x\exp\bigl(H(xy,y)\bigr)$.

(c) Show that

$$D(x) = \frac{xT'(x)T(x)}{1-T(x)} = \left(\frac{T(x)}{1-T(x)}\right)^2,$$

where $T(x)$ is the EGF for rooted labeled tree by vertices.

(d) Use Lagrange inversion with $g(u) = \left(\frac{u}{1-u}\right)^2$ to show that

$$\mu(n) = \frac{2\,(n-1)!}{n^n}\sum_{k=0}^{n-2}\binom{k+2}{2}\frac{n^{n-k-2}}{(n-k-2)!}.$$

One can obtain estimates of $\mu(n)$ for large n from this formula, but we will not do so.

11.2.17. A *functional digraph* is a simple digraph in which each vertex has outdegree 1. It is connected if the associated graph is connected. Let $\mathcal{F}_n$ be the set of connected n-vertex functional digraphs and let $f_n = |\mathcal{F}_n|$.

(a) Define a function φ from $\underline{n}^{\underline{n}}$ to digraphs with $V = \underline{n}$ as follows:

$$\text{for } g \in \underline{n}^{\underline{n}},\ \ \varphi(g) \ = \ (V,E) \ \text{ where } \ E \ = \ \{(x, g(x)) \mid x \in \underline{n}\}.$$

Prove that φ is a bijection from $\underline{n}^{\underline{n}}$ to the set of all functional digraphs with vertex set $\underline{n}$. (See Example 5.9 (p. 142).)

(b) Show that a connected functional digraph consists of a circular list of rooted trees, with each tree's edges directed toward the root and with the roots joined in a cycle (as indicated by the circular list). We're not asking for a proof, just a reasonable explanation of why this is true. Drawing some pictures may help.

(c) Show that $\sum_n f_n x^n/n! = -\ln(1-T(x))$, where $T(x)$ is the EGF for rooted labeled trees.

(d) Using the fact that $T(x) = xe^{T(x)}$ and Lagrange inversion, deduce

$$f_n \ = \ (n-1)!\sum_{k=0}^{n-1}\frac{n^k}{k!}.$$

11.2.18. A *rooted map* is an unlabeled graph that has been embedded in the plane and has had one edge distinguished by assigning a direction to it and selecting a side of it. Tutte developed some clever techniques for counting such structures. Using them it can be shown that

$$M(x) \;=\; (1-4u)(1-3u)^{-2} \quad \text{where} \quad x = u(1-3u)$$

and M is the ordinary generating function for m_n, the number of n-edge rooted maps. Prove that

$$m_n \;=\; \frac{2(2n)!\,3^n}{n!\,(n+2)!}.$$

Hint. The notation may be a bit confusing for using Theorem 11.6. Note that $T(x)$ is simply the function that is given implicitly, so in this case $T(x) = u$.

11.3 Symmetries and Pólya's Theorem

In this section we will discuss a generalization of the Burnside Lemma. We will then consider an important special case of this generalization, namely Pólya's Theorem. You should review the statement and proof of the Burnside Lemma (Theorem 4.5 (p. 112)).

Let S be a set with a permutation group G. Recall that we say $x, y \in S$ are equivalent if $y = g(x)$ for some $g \in G$. (These equivalence classes are referred to as *orbits* of G in S.) Suppose further that there is a function W defined on S such that W is constant on equivalence classes. This means that if x and y are equivalent, then $W(y) = W(x)$. We can rephrase "W is constant on equivalence classes" as "$W(g(x)) = W(x)$ for all $g \in G$ and all $x \in S$."

You may have noticed that W is not completely specified because we haven't defined its range. We don't really care what the range is as long as addition of range elements and multiplication of them by rationals is possible. Thus the range might be the real numbers, polynomials with rational coefficients, or lots of other things.

Let $\mathcal{E}$ be the set of equivalence classes of S with respect to the group G. (Don't confuse $\mathcal{E}$ with our notation for exponential generating functions.) If $B \in \mathcal{E}$, define $W(B) = W(y)$, where y is any element of B. This definition makes sense because W is constant on equivalence classes.

Theorem 11.7 The Weighted Burnside Lemma

With the above definitions,

$$\sum_{B \in \mathcal{E}} W(B) \;=\; \frac{1}{|G|} \sum_{g \in G} N(g),$$

where $N(g)$ is the sum of $W(x)$ over all $x \in S$ such that $g(x) = x$.

Before reading further, you should be able to see that the case $W(x) = 1$ for all $x \in S$ is just Burnside's Lemma. This is simply a matter of understanding the notation we've introduced.

Proof: The proof of this result is a simple modification of the proof of Burnside's Lemma. Here it is. You should be able to supply the reasons for all the steps by referring back to the proof of Burnside's Lemma.

$$\begin{aligned}
\sum_{B\in\mathcal{E}} W(B) &= \sum_{B\in\mathcal{E}}\sum_{y\in B}\frac{W(B)}{|B|} = \sum_{B\in\mathcal{E}}\sum_{y\in B}\frac{W(y)}{|B_y|} \\
&= \sum_{y\in S}\frac{|I_y|}{|G|}W(y) = \frac{1}{|G|}\sum_{y\in S}\left(\sum_{g\in G}\chi(g(y)=y)W(y)\right) \\
&= \frac{1}{|G|}\sum_{g\in G}\left(\sum_{y\in S}\chi(g(y)=y)W(y)\right) \\
&= \frac{1}{|G|}\sum_{g\in G}N(g). \ \square
\end{aligned}$$

Example 11.15 The Ferris wheel Let's return to the "Ferris wheel" problem of Example 1.12 (p. 13). Recall that we want to look at circular arrangements of ones and twos where the circles contain six digits. In this case, the group G contains 6 elements, which can be described by how they rearrange the six positions on the Ferris wheel. We called the group elements g_0 through g_5, where g_i shifts things circularly i positions.

As already noted, if we set $W(x) = 1$ for all x, then we simply end up counting all equivalence classes. Suppose, instead, that we set $W(x) = 1$ if x contains exactly 4 ones and $W(x) = 0$ otherwise. This is an acceptable definition of W because two equivalent sequences contain the same number of ones. In this case, we end up counting the equivalence classes of sequences that contain exactly 4 ones.

A more interesting example is obtained by letting z be a variable and setting $W(x) = z^k$, where k is the number of ones in x. In this case the coefficient of z^k in $W(\mathcal{E}) = \sum_{B\in\mathcal{E}} W(B)$ is the number of equivalence classes whose sequences each contain exactly k ones. In other words, $W(\mathcal{E})$ is a generating function for equivalence classes of sequences by number of ones. Let's compute $W(\mathcal{E})$ in this case. For $N(g_0)$, any sequence is allowed since $g_0(x) = x$ for all x. Thus each position can be either a two or a one. By the Rules of Sum and Product for generating functions, $N(g_0) = (1+z)^6$. If $g_1(x) = x$, all positions must be the same and so $N(g_1) = 1 + z^6$. If $g_2(x) = x$, the even numbered positions have the same value and the odd numbered positions must have the same value. By the Rules of Sum and Product for generating functions, $N(g_2) = (1+z^3)^2$. Similarly, $N(g_3) = (1+z^2)^3$, $N(g_4) = N(g_2)$ and $N(g_5) = N(g_1)$. Thus we obtain

$$\begin{aligned}
W(\mathcal{E}) &= \frac{1}{6}\left((1+z)^6 + 2(1+z^6) + 2(1+z^3)^2 + (1+z^2)^3\right) \\
&= 1 + z + 3z^2 + 4z^3 + 3z^4 + z^5 + z^6.
\end{aligned} \tag{11.25}$$

One does not need this machinery to compute the generating function. It is quite simple to construct it from the leaves of the decision tree in Figure 4.2. $\square$

We now describe a particularly important case of the Weighted Burnside Lemma. Let A and B be finite sets, let S be the functions B^A and let G be a permutation group on A. We make G into a permutation group on B^A by defining $g(f)$ for every $g \in G$ and every $f \in B^A$ to be the function given by

$$(g(f))(a) = f(g(a)) \quad \text{for all } a \in A.$$

Let W be a function from B to a set in which we can divide by integers, add and multiply. Among such sets are the set of real numbers, the set of polynomials in any number of variables, and the set

of power series. (There will be a concrete example soon.) We make W into a weight function on B^A by setting

$$W(f) = \prod_{a \in A} W(f(a)).$$

Why is W constant on equivalence classes? Since f and $g(f)$ are equivalent, the second line of $g(f)$ in two line form is simply a permutation of the second line of f; however, $W(f)$ is simply the product of the weights of the elements in its second line, without regard to their order.

Example 11.16 The Ferris wheel revisited Let's return to the previous example. We can phrase it in our new terminology:

$$\begin{aligned} A &= \{1,2,3,4,5,6\}; \\ B &= \{1,2\}; \\ G &= \text{the circular shifts of 1,2,3,4,5,6;} \\ W &= \left(\begin{smallmatrix} 1 & 2 \\ z & 1 \end{smallmatrix}\right). \end{aligned}$$

For example, if $g = (1,3,5)(2,4,6)$ and $f = \left(\begin{smallmatrix} 1&2&3&4&5&6 \\ 1&2&2&1&2&1 \end{smallmatrix}\right)$, then $g(f) = \left(\begin{smallmatrix} 1&2&3&4&5&6 \\ 2&2&1&1&1&2 \end{smallmatrix}\right)$ and $W(f) = z^3$. You should be able to verify the following observations.

(a) An element of B^A can be viewed as a 6-long sequence of ones and twos.

(b) G permutes these 6-long sequences just as the G in the previous example did.

(c) $W(f)$ is z raised to a power which equals the number of ones in the sequence $f(1), \ldots, f(6)$.

These observations show that this problem is the same as the previous one. □

Why is this special situation with $S = B^A$ important? First, because many problems that are done with the Weighted Burnside Lemma can be phrased this way. Second, because it is easier to apply the lemma in this particular case. The method for applying the lemma is known as Pólya's Theorem. Before stating the theorem, we'll look at a special case.

Example 11.17 The Ferris wheel revisited In order to compute $N(g)$ for the Ferris wheel, we need to study those functions f such that $g(f) = f$. Look at $g = (1,3,5)(2,4,6)$ again. You should be able to see that $(g(f))(a) = f(a)$ for all $a \in A$ if and only $f(1) = f(3) = f(5)$ and $f(2) = f(4) = f(6)$. For example $(g(f))(1) = f(g(1)) = f(3)$, and so $g(f) = f$ implies that $f(3) = f(1)$. More generally, you should be able to see that for any permutation g, we have $g(f) = f$ if and only if f is constant on each of the cycles of g.

To compute the sum of the weights of the functions f with $g(f) = f$, we can look at how to construct such a function:

1. First choose a value for f on the cycle $(1,3,5)$ AND
2. then choose a value for f on the cycle $(2,4,6)$.

On the first cycle, the value of f is either one OR two. If f is one, this part of f contributes z^3 to the weight $W(f)$. If f is two, this part of f contributes 1 to the weight $W(f)$. Using the Rules of Sum and Product, we get that all f with $g(f) = f$ contribute a total of $(z^3+1)(z^3+1)$; that is, $N(g) = (z^3+1)^2$. □

In order to state Pólya's Theorem, we must define the *cycle index* Z_G of a group G of permutations. Let $c_i(g)$ be the number of cycles of g of length i. Then

$$Z_G(x_1, x_2, \ldots) \;=\; \frac{1}{|G|} \sum_{g \in G} {x_1}^{c_1(g)} {x_2}^{c_2(g)} \cdots .$$

If G is the cyclic shifts of the sequence $1, 2, 3, 4, 5, 6$, you should be able to show that

$$Z_G \;=\; \frac{1}{6}\Big({x_1}^6 + 2x_6 + 2{x_3}^2 + {x_2}^3 \Big). \tag{11.26}$$

Theorem 11.8 Pólya's Theorem *If $S = B^A$ and G and W are defined as above, then*

$$\sum_{E \in \mathcal{E}} W(E) \;=\; Z_G(x_1, x_2, \ldots),$$

where

$$x_i \;=\; \sum_{b \in B} (W(b))^i.$$

This can be proved using the idea that we introduced in the previous example together with the Weighted Burnside Lemma. The proof is left as an exercise. You will probably find it easier to prove after reading some examples.

Example 11.18 The Ferris wheel revisited Now we'll apply Pólya's Theorem to derive (11.25). Since $B = \{1, 2\}$ and $W(B) = \left(\begin{smallmatrix} 1 & 2 \\ z & 1 \end{smallmatrix}\right)$, we have $x_1 = z+1$, $x_2 = z^2+1$ and so on. Substituting these values into (11.26), we obtain (11.25).

Now let's consider a Ferris wheel of arbitrary length n. Our group consists of the n cyclic shifts $g_0, \ldots, g_{n-1}$, where g_i shifts the sequence circularly i positions. This group is known as the *cyclic group* of order n and is usually denoted C_n. In order to apply Pólya's Theorem, we need to compute Z_{C_n}, which we now do.

The element g_i shifts something in position p to position $p + i$, then to $p + 2i$ and so on, where all these values are reduced modulo n, which means we divide them by n and keep the remainder. For example, if $n = 6$, $i = 4$ and $p = 3$, the successive values of the positions are 3, 1 (the remainder of 7/6), 5 and back to 3. You should be able to see easily that the length of the cycle containing p depends only on n and g_i and not on the choice of p. Thus all cycles of g_i have the same length. What is that length?

Suppose we return to position p after k steps. This can happen if and only if dividing $p + ki$ by n gives a remainder of p. In other words, ki must be a multiple of n. Since ki is also a multiple of i, it must be a multiple of the least common multiple of i and n, which is written $\operatorname{lcm}(i, n)$. Thus, the smallest possible value of ki is $\operatorname{lcm}(i, n)$. It follows that

$$\text{the length of each cycle of } g_i \text{ is } \frac{\operatorname{lcm}(i, n)}{i}. \tag{11.27}$$

Since the cycles must contain n items between all of them, the number of cycles is

$$\frac{n}{\operatorname{lcm}(i, n)/i} \;=\; \frac{ni}{\operatorname{lcm}(i, n)} \;=\; \gcd(i, n),$$

where the last equality is a fairly easy number theory result (which is left as an exercise). Incidentally, this number theory result enables us to rewrite (11.27) as

$$\text{the length of each cycle of } g_i \text{ is } \frac{n}{\gcd(i, n)}.$$

It follows from all this discussion, that g_i contributes the term $(x_{n/k})^k$, where $k = \gcd(n, i)$, to the sum for the cycle index of C_n. Thus

$$Z_{C_n} = \frac{1}{n}\sum_{i=0}^{n-1}(x_{n/\gcd(n,i)})^{\gcd(n,i)}. \qquad 11.28$$

With $n = 6$, (11.28) gives us (11.26), as it should. Carry out the calculation. Notice that some of the terms were equal so we were able to collect them together. It would be nice to do that in general. This means we need to determine when various values of $\gcd(n, i)$ occur. We leave this as an exercise. □

Example 11.19 Unlabeled rooted trees Although we have studied the number of various types of unlabeled RP-trees, we haven't counted those that are not in the plane. There's a good reason for that—we need Pólya's theorem or something similar.

We'll look at unlabeled rooted trees where each vertex has at most three edges directed away from the root. Let t_n be the number with n vertices. We want to study $T(x)$, the ordinary generating function for the sequence $t_0, t_1, \ldots$. There are various reasons for doing this. First, it is not too difficult and not too easy. Second, these trees correspond to an important class of organic compounds: Each of the vertices corresponds to a carbon atom. These carbon atoms are all joined by single bonds. A "radical" (for example, –OH or –COOH) is attached by a single bond to the carbon atom that corresponds to the root Finally, to give each carbon atom valency four, hydrogen atoms are attached as needed. Compounds like this with the –OH radical are known as *alcohols*. Two alcohols with the same number of carbon atoms but different associated trees are called *isomers*. The two isomers of propyl alcohol are

The corresponding rooted trees are ○—●—○ and ○—○—●, respectively, where ● indicates a root and ○ a nonroot.

We can approach this problem the same way we did RP-trees: We take a collection of unlabeled rooted trees and join them to a new root. There are two important differences from our previous considerations of RP-trees.

- Since a vertex has at most three sons, we must take this into account. (Previously we dealt mostly with exactly two sons.)
- There is *no ordering among the sons*. This is what we mean by not being in the plane—the sons are not ordered from left to right; they are simply a multiset of trees. In terms of symmetries, this means that all permutations of the sons give equivalent trees.

Let's begin with the problem of at most three sons. One way to handle this is to sum up the cases of exactly one, two and three sons. There is an easier way. We will allow the empty tree, so $t_0 = 1$. By taking three trees, we get at most three sons since any or all of them could be the empty tree.

Pólya's Theorem can be applied in this situation. In the notation of the theorem, $A = \underline{3}$ and B is the set of rooted trees that we are counting. A function in B^A selects a list of three trees to be the sons of the root. Since all permutations of these sons are possible, we want to study the group of all possible permutations on three things. This group is known as the symmetric group on three things and is denoted S_3. More generally, one can speak of S_n. Since all permutations of n things are allowed, S_n contains $n!$ elements, the number of permutations of an n-set.

By writing down all six permutations of $\underline{3}$, you should be able to easily show that

$$Z_{S_3} = \frac{x_1^3 + 3x_1x_2 + 2x_3}{6}.$$

We need to compute x_i so that we can apply Pólya's Theorem. As noted earlier, B is the set of all unlabeled rooted trees of the sort we are constructing. $W(b)$ for a tree b is simply x^k, where k is the number of vertices of b. It follows that $x_i = T(x^i)$. Thus we have

$$T(x) \;=\; 1 + x\frac{T(x)^3 + 3T(x)T(x^2) + 2T(x^3)}{6}, \tag{11.29}$$

where the "1+" is present because a (possibly empty) tree is constructed by taking the empty set OR ....

Equation (11.29) can be used to compute t_n recursively. You should be able to do this. If you do this for n up to five or so, you will discover that it is probably easier to simply list the trees and then count them. On the other hand, suppose you want the answer up to $n = 20$. You will probably want to use a computer. While it is certainly possible to write a program to list the trees, it is probably easier to use a symbolic manipulation package. Simply start with $T(x) = 1$, which is obviously the beginning of the generating function. Then apply (11.29) n times to compute new values of $T(x)$. To avoid overflow and/or excessive running time, you should truncate all calculations to terms of degree at most n.

There is another situation in which (11.29) is better than listing. Suppose that we want to get an estimate of, say t_{100}. Asymptotic methods provide a way for doing this using (11.29). See Example 11.34 (p. 354). □

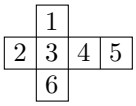
In general, computing the cycle index of a group is not a simple matter. The examples considered so far have involved relatively simple groups. The following example deals with a somewhat more complicated situation—the cycle index of the group of symmetries of a cube, where the group is a permutation group on the faces of the cube.

Example 11.20 Symmetries of the cube

In Exercise 4.2.7 (p. 110), you used a decision tree to study the ways to color the faces of a cube. Here we'll use a cycle index polynomial to study it. Before doing this, we must answer two questions:

- What symmetries are possible? Certainly, we should allow the rotations of the cube. There are other symmetries that involve reflections. Whether we allow these or not will depend on the problem at hand. A solid cube in the real world can only be rotated; however, reflections of a cube that is associated with a chemical compound (like the trees in the previous example) may be a perfectly acceptable physical manipulation. We'll begin with just rotations and then allow reflections as well.
- What objects are being permuted? Obvious choices are the vertices, edges and faces of the cube. There are less obvious ones such as diagonals. Different choices for the objects will, in general, lead to different permutation groups and hence different cycle index polynomials. Since we are coloring faces, we'll choose the faces of the cube and leave other objects as exercises.

Before proceeding, we recommend that you find a cube if you can. Possibilities include a sugar cube, a die and a homemade cube consisting of six squares of cardboard taped together. Here is a picture of an "unfolded" cube.



11.30

If you imagine the square marked 3 as being the base, squares 1, 2, 4 and 6 fold up to produce sides and square 5 folds over to become the top.

What axes can the cube be rotated around and through what angles so that it will occupy the same space? The axes fall into three classes:

- Face centered: This type goes through the centers of opposite pairs of faces. There are three of them, through the faces 1-6, 2-4 and 3-5. A cube can be rotated 0°, ±90° or 180° about such an axis.

- Edge centered: This type goes through the centers of opposite pairs of edges. There are six of them. An edge can be described by the two faces that lie on either side of it. In this notation, one edge centered axis is 25-34. A cube can be rotated 0° or 180° about such an axis.
- Vertex centered: This type goes through opposite vertices of the cube. There are four of them, one of which is 125-346, where a vertex is described by the three faces that meet there. A cube can be rotated 0° or ±120° about such an axis.

To compute a term in the cycle index polynomial corresponding to a rotation, we can look at how the rotation permutes the faces and then determine the term from the cycle lengths. For example, the face centered rotation about 1-6 through 90° gives the permutation $(1)(2,3,4,5)(6)$. This gives us the term $x_1x_4x_1 = x_1^2x_4$. You should be able to establish the following terms by studying your cube. The letters F, E and V describe the type of axis.

no rotation	x_1^6
±90° F	$x_1^2x_4$
180° F	$x_1^2x_2^2$
180° E	x_2^3
±120° V	x_3^2

Altogether there are 24 rotations. (We have counted the rotations through 0° just once.)

Before we add up the 24 terms, we must verify that the rotations are all distinct. One way to do this is by looking at the cycles of the 24 permutations of the faces and noting that they are distinct. Can you think of another method for seeing that they are distinct? You might try a different approach—instead of showing that the rotations are distinct directly, show that there must be 24 distinct rotations by a geometric argument and then use the fact that we have found all possible rotations.

Adding up the 24 terms, the cycle index polynomial for the rotations of the cube in terms of the faces of the cube is

$$\frac{x_1^6 + 6x_1^2x_4 + 3x_1^2x_2^2 + 6x_2^3 + 8x_3^2}{24}. \tag{11.31}$$

It follows that the number of rotationally inequivalent ways to color the faces of a cube using k colors is

$$C(k) \;=\; \frac{k^6 + 6k^3 + 3k^4 + 6k^3 + 8k^2}{24} \;=\; \frac{k^6 + 3k^4 + 12k^3 + 8k^2}{24}. \tag{11.32}$$

How many rotationally inequivalent ways can the cube be colored with 3 colors so that every color appears? We cannot substitute directly into the cycle index polynomial, but we can see the answer with a little thought. Can you do it?

* * * Stop and think about this! * * *

One solution is to use the Principle of Inclusion and Exclusion together with (11.32). The answer is

$$C(3) - 3C(2) + 3C(1) - C(0) \;=\; 57 - 3 \times 10 + 3 \times 1 - 0 \;=\; 30.$$

How many rotationally inequivalent ways can we color the faces of a cube with k colors so that adjacent faces have distinct colors? This problem cannot be answered with Pólya's Theorem; however, it can be done with Burnside's Lemma.

We now turn our attention to rotations and reflections of the cube. Imagine the cube made from (11.30) has been rotated in any fashion. Now carry out the following operations.

- Rotate it so that face 3 is on the bottom and face 2 is on the left.
- Open the cube by cutting two sides of face 4 and all sides of face 5 except the one between it and face 4.

This will always lead to the picture in (11.30). Now suppose you do the same thing with a cube that has been rotated and, possibly, reflected. The result will be either (11.30) or (11.30) with 1 and 6 interchanged. You should convince yourself of this by experimentation or by geometric arguments.

The result of the previous paragraph implies that any reflection and rotation combination can be obtained as either a rotation or a rotation followed by an interchange of the labels 1 and 6. Using this observation, the cycle index of the group of rotations and reflections can be computed from a knowledge of the orbits of the rotations. We will not carry out the tedious details. The result is

$$\frac{x_1^6 + 3x_1^4x_2 + 6x_1^2x_4 + 9x_1^2x_2^2 + 7x_2^3 + 6x_2x_4 + 8x_3^2 + 8x_6}{48}$$

There are alternative geometric arguments that could be used to obtain this result. For example, one can look at the possible arrangement of faces around the upper left front corner of the cube. □

*Example 11.21 Counting unlabeled graphs How many unlabeled graphs are there with n vertices and q edges? This has many variants: do we allow multiple edges? loops? Are the graphs directed?

All of these can be done in a similar manner and all lead to messy expressions. We'll choose the simplest case: simple directed graphs with loops allowed.

In any of these situations, we use Pólya's Theorem. Suppose that n, the number of vertices, is given. Let $\underline{n}$ be the set of vertices. The functions we consider will be from $\underline{n} \times \underline{n}$ to $\{0, 1\}$. The value of $f((u, v))$ is the number of edges from u to v. In the notation of Pólya's Theorem, $S = B^A$ and G acts on A. We have already said that $B = \{0, 1\}$ and $A = \underline{n} \times \underline{n}$. What is G and what is the weight W? The group G will be the group of all permutations of n things, but instead of acting on the vertices $\underline{n}$, it must act on the ordered pairs $\underline{n} \times \underline{n}$. Most of this example will be devoted to explaining and computing the cycle index of this group action. W is given by $W(i) = y^i$. The coefficient of y^q will then be the number of unlabeled simple digraphs with n vertices and q edges.

Before turning to the calculations, we remark how some of the other graph counting problems can be dealt with. If loops are not allowed, remove the n ordered pairs of the form (i, i) from A. If any number of edges is allowed, replace B by the nonnegative integers, still setting $W(i) = y^i$. To count (loopless) graphs, replace $\underline{n} \times \underline{n}$ with $\mathcal{P}_2(\underline{n})$, the set of 2 element subsets of $\underline{n}$.

Let g be a permutation acting on $\underline{n}$. If we write g in cycle form, it is fairly easy to translate g into a permutation of $\underline{n} \times \underline{n}$. For example, suppose that $n = 3$ and $g = (1, 2)(3)$. To avoid confusing ordered pairs with cycles, we will indicate an ordered pair without parentheses and commas, e.g., 13 instead of $(1, 3)$. Using g, we have the following two line form for the corresponding permutation of $\underline{3} \times \underline{3}$

$$\begin{pmatrix} 11 & 12 & 13 & 21 & 22 & 23 & 31 & 32 & 33 \\ 22 & 21 & 23 & 12 & 11 & 13 & 32 & 31 & 33 \end{pmatrix},$$

which you should be able to verify easily. In cycle form this is

$$(11, 22)(12, 21)(13, 23)(31, 32)(33),$$

which contributes $x_1x_2^4$ to the cycle index. How do we do this in general?

Suppose $u, v \in \underline{n}$ are two vertices, that u belongs to a cycle of g of length i and that v belongs to a cycle of length j. The length of the cycle containing uv is the number of times we must apply g in order to return to uv. After we apply g to uv k times, we will have advanced k positions in the cycle containing u and k positions in the cycle containing v. Thus, we will return to uv after k times if and only if k is a multiple of i and k is a multiple of j. The smallest positive such k is $\operatorname{lcm}(i, j)$, the least common multiple of i and j. Thus uv belongs to a cycle of length $\operatorname{lcm}(i, j)$ and this cycle contributes a factor of $x_{\operatorname{lcm}(i,j)}$ to a term of the cycle index.

Let's look more closely at the set of directed edges st that can be formed by choosing s from the same cycle as u and t from the same cycle as y. There are ij such edges. Like uv, each edge st lies in a cycle of length $\operatorname{lcm}(i, j)$. Thus, there are $ij/\operatorname{lcm}(i, j) = \gcd(i, j)$ such cycles.

Let's look carefully at what we have shown. If we choose a cycle C of length i and another cycle D of length j, then the ij ordered pairs in $C \times D$ lie in $\gcd(i,j)$ cycles of length $\operatorname{lcm}(i,j)$. Thus they contribute a factor of $x_{\operatorname{lcm}(i,j)}^{\gcd(i,j)}$ to a term of the cycle index.

If g acting on $\underline{n}$ has exactly ν_k cycles of length k, the argument in the previous paragraph shows that it contributes the term

$$\prod_i \prod_j \Big(\big(x_{\operatorname{lcm}(i,j)}\big)^{\gcd(i,j)} \Big)^{\nu_i \nu_j}$$

to the cycle index we are computing. This gives us the following recipe for computing the cycle index.

Theorem 11.9 *To compute the cycle index for the n-vertex unlabeled digraphs (with loops allowed), start with the cycle index for the set of all $n!$ permutations of $\underline{n}$. Replace every term*

$$c_i \prod_{i=1}^{n} x_i^{\nu_i} \quad \textit{with} \quad c_i \prod_{i,j} \big(x_{\operatorname{lcm}(i,j)}\big)^{\nu_i \nu_j \gcd(i,j)},$$

where the latter product extends over all (i,j). □

Exercises

11.3.1. This deals with the cycle index of C_n, the cyclic group. You will need to know that, up to the order of the factors, every number can be factored uniquely as a product of primes. We take that as given.

(a) Suppose that $A = p_1^{a_1} \cdots p_k^{a_k}$ where $p_1, \ldots, p_k$ are distinct primes. We use the shorthand notation $A = \mathbf{p}^{\mathbf{a}}$ for this product. Suppose that $B = \mathbf{p}^{\mathbf{b}}$. Let $c_i = \min(a_i, b_i)$, $d_i = \max(a_i, b_i)$, $C = \mathbf{p}^{\mathbf{c}}$ and $D = \mathbf{p}^{\mathbf{d}}$. Prove that $AB = CD$, $C = \gcd(A,B)$ and $D = \operatorname{lcm}(A,B)$. This establishes the claims in Example 11.18.

(b) Prove that the number of integers i in $\{1, \ldots, n\}$ for which $\gcd(n,i) = k$ is

- zero if k does not divide n;
- the number of integers j in $\{1, \ldots, \frac{n}{k}\}$ for which $\gcd(j, n/k) = 1$ if k divides n. This latter number is denoted $\varphi(n/k)$ and is called the *Euler phi function*. We discussed how to compute it in Exercise 4.1.5 (p. 100).

(c) Conclude that

$$Z_{C_n} = \frac{1}{n} \sum \varphi(n/k) z_{n/k}^{k} = \frac{1}{n} \sum \varphi(k) z_k^{n/k},$$

where the sum ranges over all integers between 1 and n inclusive that divide n.

11.3.2. The following questions refer to the group of rotations of the cube.

(a) Compute the cycle index of the group acting on the edges of the cube.

(b) Compute the cycle index of the group acting on the vertices of the cube.

(c) Imagine three perpendicular axes drawn through the center of the cube joining the centers of faces. Label the axes x, y and z, but *do not distinguish a direction on the axes*—thus the axes are simply lines. Compute the cycle index of the group acting on these axes.

(d) Repeat the previous question where a direction is assigned to each of the axes. Reversal of the direction of an axis is indicated by a minus sign; e.g., a rotation that reverses the z-axis and interchanges the x-axis and y-axis is written in cycle notation as

$$(x,y)(-x,-y)(z,-z).$$

11.3.3. The regular octahedron consists of eight equilateral triangles joined together so that the result looks like two pyramids joined base to base. A regular octahedron can be obtained by placing a vertex in each face of a cube and joining two vertices if they lie in faces which are separated by an edge.

(a) There is a duality between a regular octahedron and a cube in which faces correspond to vertices, edges to edges and vertices to faces. Obtain this correspondence.

(b) Write down the cycle index for the group of symmetries of the regular octahedron (reflections allowed or not) acting on the vertices of the regular octahedron.
Hint. This requires no calculation on your part.

(c) Do the same for the rotations of the octahedron acting on the edges.

11.3.4. Write down the cycle index for the group of rotations of the regular tetrahedron acting simultaneously on the vertices, edges and faces. Instead of the usual x_i, use v_i, e_i and f_i, indicating whether it is an orbit of vertices, faces or edges. For example, the identity rotation gives the term $v_1^4 e_1^6 f_1^4$. Explain how to use this result to obtain the cycle index for the group acting on just the edges.

11.3.5. Write down the cycle index polynomial for all permutations of $\underline{4}$ and use this to write down the ordinary generating function $D_4(y)$ for simple unlabeled 4-vertex digraphs by number of edges.

11.3.6. Repeat the previous exercise with "4" replaced by "5." You may find Exercises 2.2.5 and 2.3.3 (p. 57) helpful.

*11.3.7. State and prove a theorem like Theorem 11.9 for unlabeled n-vertex simple (loopless) graphs.
Hint. You will need to distinguish two cases depending on whether or not u and v are in the same cycle of g acting on $\underline{n}$.

11.4 Asymptotic Estimates

The area of asymptotics deals with obtaining estimates for functions for large values of the variables and, sometimes, for values near zero. Since the domain of the functions we're concerned with is the positive integers, these functions can be thought of as sequences $a_1, a_2, \ldots$. Since this section uses the terminology introduced in Appendix B, you may want to review it at this time.

A solid mathematical treatment of asymptotics requires more background than we are willing to assume and developing the background would take too much time. Therefore, the material in this section is not rigorous. Instead, we present several principles which indicate what the result will almost certainly be in common combinatorial situations. The intent of this section is to give you a feeling for the subject, some direction for future study and some useful rules of thumb.

Before launching into specific tools and examples, we'd like to set the stage a bit since you are probably unfamiliar with asymptotic estimates. The lack of specific examples may make some of this introductory material a bit vague, so you may want to reread it after completing the various subsections.

Suppose we are interested in a sequence of numbers. We have four methods of providing asymptotic information about the numbers. Here they are, with examples:

- A combinatorial description: say B_n is the number of partitions of an n-set;
- A recursion: $F_0 = 1$, $F_1 = 2$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$;
- A formula: the number of involutions of an n-set is

$$\sum_{j=0}^{n} \frac{n!}{j!\, 2^j (n-2j)!};$$

the number of unlabeled full binary RP-trees with n leaves is $\frac{1}{n}\binom{2n-2}{n-1}$;

- A generating function: the ordinary generating function for the number of comparisons needed to Quicksort an n long list is
$$\frac{-2\ln(1-x)-2x}{(1-x)^2};$$
the ordinary generating function for the number of unlabeled rooted full binary trees by number of leaves satisfies
$$T(x) = \frac{T(x)^2+T(x^2)}{2}+x.$$

Given such information, can we obtain some information about the size of the terms in the sequence? The answer will, of course, depend on the information we are given. Here is a quick run down on the answers.

- A combinatorial description: It is usually difficult, if not impossible, to obtain information directly from such a description.
- A recursion: It is often possible to obtain some information. We will briefly discuss a simple case.
- A formula: The formula by itself may be explicit enough. If it is not, using Stirling's formula may suffice. If a summation is present, it can probably be estimated if all its terms are nonnegative, but it may be difficult or impossible to estimate a sum whose terms alternate in sign. Unfortunately, the estimation procedures usually involve a fair bit of messy calculation and estimation. We will discuss two common types of sums.
- A generating function: If the generating function converges for some values of x other than $x = 0$, it is quite likely that estimates for the coefficients can be obtained by using tools from analysis. Tools have been developed that can be applied fairly easily to some common situations, but rigorous application requires a background in complex analysis. The main emphasis of this section is the discussion of some simple tools for generating functions.

You may have noticed that we have discussed only singly indexed sequences in our examples. There are fewer tools available for multiply indexed sequences and they are generally harder to describe and to use. Therefore, we limit our attention to singly indexed sequences.

There is no single right answer to the problem of this section—finding simple approximations to some a_n for large n—we must first ask how simple and how accurate.

We will not try to specify what constitutes a simple expression; however, you should have some feel for it. For example, an expression of the form an^bn^c, where a, b and c are constants, is simple. The expression $\sqrt{2\pi n}(n/e)^n$ is simpler than the expression $n!$ even though the latter is more easily written down. Why? If we limit ourselves to the basic operations of addition, subtraction multiplication, division and exponentiation, then the former expression requires only six operations while $n!$ requires $n-1$ multiplications. We have hidden the work of multiplication by the use of a function, namely the factorial function. We can estimate simplicity by counting the number of basic operations.

There are wide variations in the degree of accuracy that we might ask for. Generally speaking, we would like an approximating expression whose relative error goes to zero. In other words, given a_n, we would like to find a simple expression $f(n)$ such that $a_n/f(n) \to 1$ as $n \to \infty$. In this case we say that a_n is *asymptotic to* $f(n)$ and write $a_n \sim f(n)$. Sometimes greater accuracy is desired—a problem we will not deal with. Sometimes we may have to settle for less accuracy—a situation we will be faced with.

The discussion of accuracy in the previous paragraph is a bit deceptive. What does $a_n \sim f(n)$ tell us about specific values? Nothing! It says that *eventually* $a_n/f(n)$ gets as close to 1 as we may desire, but eventually can be a *very* long time. But, in most cases of interest, we are lucky enough that the ratio $a_n/f(n)$ approaches 1 fairly quickly. Can we be more precise?

It is possible in most cases to compute an upper bound on how slowly $a_n/f(n)$ approaches 1; that is upper bounds on $|a_n/f(n) - 1|$ as a function of n. Obtaining such bounds often involves a considerable amount of work and is beyond the scope of this text. Even if one has a bound it may be unduly pessimistic—the ratio may approach one much faster than the bound says. A more pragmatic approach is to compute $a_n/f(n)$ for small values of n and hope that the trend continues for larger values. Although far from rigorous, this pragmatic approach almost always works well in practice. We'll carry out such calculations for the problems studied in this section.

The following subsections are independent of each other. If you are only going to read one of them, we recommend the one on generating functions.

Recursions

We have been able to solve the simplest sorts of recursions in earlier sections. Now our interest is different—we want asymptotic information from the recursions. We will consider two types of linear recursions that arise frequently in the analysis of algorithms. A *linear recursion* for a_n is an expression of the form

$$a_n = c_1(n)a_{n-1} + c_2(n)a_{n-2} + \ldots + c_n(n)a_0 + f(n)$$

for $n \geq N$. If $f(n) = 0$ for $n \geq N$, the recursion is called *homogeneous.*

We first discuss homogeneous recursions whose coefficients are "almost constant." In other words, except for initial conditions,

$$a_n = c_1(n)a_{n-1} + c_2(n)a_{n-2} + \ldots + c_k(n)a_{n-k}, \tag{11.33}$$

where the functions $c_i(n)$ are nearly constant. If $c_i(n)$ is nearly equal to C_i, then the solution to

$$A_n = C_1A_{n-1} + C_2A_{n-2} + \ldots + C_kA_{n-k}, \tag{11.34}$$

with initial conditions, should be reasonably close to the sequence a_n. We will not bother to discuss what reasonably close means here. It is possible to say something about it, but the subject is not simple.

What can we say about the solution to (11.34)? Without the initial conditions, we cannot say very much with certainty; however, the following is usually true.

Principle 11.1 Constant coefficient recursions *Let r be the largest root of the equation*

$$r^k = C_1r^{k-1} + C_2r^{k-2} + \ldots + C_kr^0. \tag{11.35}$$

If this root occurs with multiplicity m, then there is usually a constant A such that the solution to (11.34) that satisfies our (unspecified) initial conditions is asymptotic to $An^{m-1}r^n$.

This result is not too difficult to prove. Given the initial conditions, one can imagine using (11.34) to obtain a rational function for $\sum_n A_nx^n$. The denominator of the rational function will be $p(x) = 1 - (C_1x + \ldots + C_kx^k)$. Now imagine expanding the result in partial fractions. The reason for the lack of a guarantee in the principle is that a factor may cancel from the numerator and denominator of $\sum_n A_nx^n$, giving a lower degree polynomial than $p(x)$.

The principle is perhaps not so interesting because it gives much less accurate results than those obtained by using generating functions and partial fractions. Its only attractions are that it requires less work and gives us an idea of what to expect for (11.33).

Principle 11.2 Linear recursions *Suppose that $c_i(n) \to C_i$ as $n \to \infty$ and that at least one C_i is nonzero. Let r be the largest root of (11.35). Then r^n is probably a fairly reasonable crude approximation to the solution a_n of (11.33) that satisfies our (unspecified) initial conditions. Usually*

$$\lim_{n\to\infty} (a_n)^{1/n} \;=\; r.$$

Example 11.22 Involutions Let a_n be the number of permutations of $\underline{n}$ which are involutions, that is, no cycle lengths exceed two. Either n is in a cycle with itself OR it forms a cycle with one of the remaining $n-1$ elements of $\underline{n}$. Thus

$$a_n \;=\; a_{n-1} + (n-1)a_{n-2},$$

with some appropriate initial conditions.

The coefficients of this recursion are not almost constant, but we can use a trick, which works whenever we have coefficients which are polynomials in n. Let $b_n = a_n/(n!)^d$, where d is to be determined. Dividing our recursion by $(n!)^d$, and doing a little simple algebra, we have

$$b_n \;=\; \frac{1}{n^d} b_{n-1} + \frac{n-1}{(n^2-n)^d} b_{n-2}.$$

If $d < 1/2$, the last coefficient is unbounded, while if $d > 1/2$, both coefficients on the right side approach 0. On the other hand, with $d = 1/2$, the first coefficient approaches 0 and the second approaches 1. Thus we are led to consider the recursion $b_n = b_{n-2}$ and hence the roots of the polynomial $r^2 = 1$. Since the largest is $r = 1$, we expect that b_n should approach 1. Thus $(n!)^{1/2}$ is a rough approximation to a_n. We can eliminate the factorial by using Stirling's formula (Theorem 1.5 (p. 12)). Since the approximation in Principle 11.2 is so crude we may as well ignore factors like $\sqrt{2\pi n}$ and simply say that a_n probably grows roughly like $(n/e)^{n/2}$. □

We now present a type of recursion that often arises in divide and conquer problems such as Mergesort. Some authors have called various forms of the theorem associated with this principle a master theorem for recursions. The reason we have put "function" in quotes is explained after the principle.

Principle 11.3 Master Principle for Recursions *We want to study the "function" $T(n)$. Suppose that for some "functions" $f(n), s_1(n), \ldots, s_w(n)$, some N, and some $0 < c < 1$ we have*

(i) $T(n) > 0$ *for* $n \geq N$,

(ii) $f(n) \geq 0$ *for* $n \geq N$,

(iii) $s_i(n) - cn \in O(1)$ *for* $1 \leq i \leq w$,

(iv) $T(n) = f(n) + T(s_1(n)) + T(s_2(n)) + \cdots + T(s_w(n))$.

Let $b = \log w / \log(1/c)$. *Then*

(a) if $f(n)/n^b \to \infty$ *as* $n \to \infty$, *then we usually have* $T(n) \in \Theta(n^b)$;

(b) if $f(n)/n^b \to 0$ *as* $n \to \infty$, *then we usually have* $T(n) \in \Theta(f(n))$;

(c) if $n^b \in \Theta(f(n))$, *then we usually have* $T(n) \in \Theta(n^b \log n)$.

The principle says that $T(n)$ grows like the faster growing of $f(n)$ and n^b unless they grow at the same rate, in which case $T(n)$ grows faster by a factor of $\log n$.

Why is "function" in quotes? Consider merge sorting. We would like $T(n)$ to be the number of comparisons needed to merge sort an n-long list. This is not a well-defined function! The number depends on the order of the items in the original list. (See Example 7.13 (p. 211).) Thus, we want to let $T(n)$ stand for the set of all possible values for the number of comparisons that are required. Hence $T(n)$ is really a collection of functions. Similarly $f(n)$ and, more rarely, the $s_i(n)$ may be collections of functions. Thus, a statement like $T(n) \in \Theta(f(n))$ should be interpreted as meaning that the statement is true for all possible values of $T(n)$ and $f(n)$.

Example 11.23 Recursive multiplication of polynomials Suppose we want to multiply two polynomials of degree at most n, say

$$P(x) = p_0 + p_1x + \cdots + p_nx^n \quad \text{and} \quad Q(x) = q_0 + q_1x + \cdots + q_nx^n.$$

A common method for doing this is to use the distributive law to generate $(n+1)^2$ products p_0q_0, p_0q_1x, $p_0q_2x^2$, …, $p_nq_nx^{2n}$ and then collect the terms that have the same powers of x. This involves $(n+1)^2$ multiplications of coefficients and, it can be shown, n^2 additions of coefficients. Thus, the amount of work is $\Theta(n^2)$.

Is this the best we can do? Of course, we can do better if $P(x)$ or $Q(x)$ have some coefficients that are zero. Since we're concerned with finding a general algorithm, we'll ignore this possibility. There is a recursive algorithm which is faster. It uses the following identity, which you should check

Identity: If $P_L(x)$, $P_H(x)$, $Q_L(x)$ and $Q_H(x)$ are polynomials, then

$$\begin{aligned}\big(P_L(x) + P_H(x)x^m\big)\big(Q_L(x) + Q_H(x)x^m\big) \\ = A(x) + \big(C(x) - A(x) - B(x)\big)x^m + B(x)x^{2m}\end{aligned}$$

where

$$A(x) = P_L(x)Q_L(x), \quad B(x) = P_H(x)Q_H(x),$$

and

$$C(x) = \big(P_L(x) + P_H(x)\big)\big(Q_L(x) + Q_H(x)\big).$$

We can think of this identity as telling us how to multiply two polynomials $P(x)$ and $Q(x)$ by splitting them into lower degree terms (the polynomials $P_L(x)$ and $Q_L(x)$) and higher degree terms (the polynomials $P_H(x)x^m$ and $Q_H(x)$):

$$P(x) = P_L(x) + P_H(x)x^m \quad \text{and} \quad Q(x) = Q_L(x) + Q_H(x)x^m.$$

The identity requires three polynomial multiplications to compute $A(x)$, $B(x)$ and $C(x)$. Since the degrees involved are only about half the degrees of the original polynomials, we've gone from about n^2 multiplications to about $3(n/2)^2 = 3n^2/4$, an improvement by a constant factor as $n \to \infty$. When this happens, applying an algorithm recursively usually gives an improvement in the exponent. In this case, we expect $\Theta(n^d)$ for some $d < 2$ instead of n^2.

Here's a recursive algorithm for multiplying two polynomials $P(x) = p_0 + p_1x \cdots + p_nx^n$ and $Q(x) = q_0 + q_1x + \cdots + q_nx^n$ of degree at most n.

```
MULT(P(x), Q(x), n)
    If (n=0)  Return p_0 q_0
    Else
        Let m = n/2 rounded up.
        P_L(x) = p_0 + p_1 x + ··· p_{m-1} x^{m-1}
        P_H(x) = p_m + p_{m+1} x + ··· p_n x^{n-m}
        Q_L(x) = q_0 + p_1 x + ··· q_{m-1} x^{m-1}
        Q_H(x) = q_m + q_{m+1} x + ··· q_n x^{n-m}
```

```
        A(x) = MULT(P_L(x), Q_L(x), m − 1)
        B(x) = MULT(P_H(x), Q_H(x), n − m)
        C(x) = MULT(P_L(x) + P_H(x), Q_L(x) + Q_H(x), n − m)
        D(x) = A(x) + (C(x) − A(x) − B(x))x^m + B(x)x^{2m}
        Return D(x)
    End if
End
```

We store a polynomial as array of coefficients. The amount of work done in calculation is the number of times we multiply or add two coefficients.

Let's count multiplications. Since a polynomial of degree n has $n+1$ coefficients (constant term to coefficient of x^n), we'll denote by $T(n+1)$ the number of multiplications for a polynomial of degree n. To multiply two constants, we have $T(1) = 1$. You should be able to show that the recursive part of the algorithm gives us $T(n) = T(m)+T(n-m)+T(n-m)$, where $m = \lfloor n/2 \rfloor$, the largest integer not exceeding $n/2$. The Master Principle applies with $w = 3$, $f(n) = 0$, $s_1(n) = \lfloor n/2 \rfloor$, $s_2(n) = s_3(n) = n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$[1] and $c = 1/2$. Thus $b = \log 3/\log 2$ and $T(n) \in \Theta(n^{\log 3/\log 2})$. Since $\log 3/\log 2$ is about 1.6 which is less than 2, this is less work than the n^2 multiplications in the common method when n is large.

What about additions and subtractions? The polynomials $A(x)$, $B(x)$ and $C(x)$ all have degree about n. Thus, computing $C(x) - A(x) - B(x)$ requires about $2n$ subtractions. The polynomials $A(x)$ and $(C(x) - A(x) - B(x))x^m$ overlap in about $n/2$ terms and so adding them together requires about $n/2$ additions. Adding in $B(x)x^{2m}$ requires about $n/2$ more additions. Thus, there are about $3n$ additions and subtractions involved in computing $D(x)$ from A, B and C. If $U(n)$ is the number of additions and subtractions in the recursive algorithm for polynomials of degree $n-1$,

$$U(n) = f(n) + U(m) = U(n-m) + U(n-m) \qquad \text{where} \qquad f(n) \in \Theta(3n).$$

By the Master Principle with $b = \log 3/\log 2$, $U(n) \in \Theta(n^b)$. Thus the algorithm requires $\Theta(n^{\log 3/\log 2})$ multiplications, additions and subtractions. □

Sums of Positive Terms

Suppose that

$$a_n = \sum_{k=0}^{L_n} t_{n,k},$$

where $t_{n,k} > 0$ and $L_n \to \infty$. Imagine n fixed and think of $t_{n,k}$ as a sequence in k; i.e., $t_{n,0}$,$t_{n,1}$, ... Let $r_k(n) = t_{n,k+1}/t_{n,k}$, the ratio of consecutive terms. Usually we simply write r_k for $r_k(n)$. In practice we usually have one of four situations

(a) **Decreasing terms** ($r_k \le 1$ for all k). We will study this.

(b) **Increasing terms** ($r_k \ge 1$ for all k) Convert to (a) by writing the sequence backwards:

$$a_n = \sum_{k=0}^{L_n} t_{n,k} = \sum_{i=0}^{L_n} t_{n,L_n-i} = \sum_{k=0}^{L_n} s_{n,k},$$

(c) **Increasing, then decreasing** ($r_k \le 1$ for $k < K_n$ and $r_k \ge 1$ for $k \ge K_n$): split the sum at K_n. This gives one sum like (a) and one like (b):

$$a_n = \sum_{k=0}^{L_n} t_{n,k} = \sum_{k=0}^{K_n-1} t_{n,k} + \sum_{k=0}^{M_n} u_{n,k},$$

[1] The ceiling function $\lceil x \rceil$ is the least integer not exceeding x.

where $M_n = L_n - K_n$ and $u_{n,k} = t_{n,k+K_n}$.

(d) **Decreasing, then increasing** ($r_k \geq 1$ for $k < K_n$ and $r_k \leq 1$ for $k \geq K_n$: Split into two as done for (c).

Suppose we are dealing with (a), decreasing terms, and that $\lim_{n\to\infty} r_k(n) = r$ exists for each k and does not depend on k. This may sound unusual, but it is quite common. If $r = 1$, we will call the terms *slowly decreasing*. If $|r| < 1$, we will call the terms *rapidly decreasing*. The two sums obtained from Case (c) are almost always slowly decreasing and asymptotically the same.

Principle 11.4 Sums of rapidly decreasing terms *If there is an r with $0 < r < 1$ such that $\lim_{n\to\infty} t_{n,k+1}/t_{n,k} = r$ for each value of k, then we usually have a geometric series approximation:*

$$\sum_{k=0}^{L_n} t_{n,k} \sim \sum_{k\geq 0} t_{n,0} r_n^k \sim \frac{t_{n,0}}{1-r}. \tag{11.36}$$

(Note that r does not depend on k.)

Aside: Actually, there is a more general principle: If $r_k(n) \to \rho_k$ as $n \to \infty$ and there is an $R < 1$ such that $|\rho_k| < R$ for all k, then the sum is asymptotic to $t_{n,0} \sum_{i=0}^{\infty} \rho_1 \cdots \rho_i$. Principle 11.4 is the special case $\rho_i = r$ for all i.

Example 11.24 Small subsets How many subsets of an n-set have at most cn elements where $c < 1/2$? You should have no trouble seeing that the answer is

$$\sum_{k=0}^{\lfloor cn \rfloor} \binom{n}{k},$$

where $\lfloor cn \rfloor$ is the largest integer that does not exceed cn. Let's approximate the sum. Since the terms are increasing, we reverse the order:

$$\sum_{k=0}^{\lfloor cn \rfloor} \binom{n}{\lfloor cn \rfloor - k}.$$

We have

$$\frac{t_{n,k+1}}{t_{n,k}} = \frac{\binom{n}{\lfloor cn \rfloor - k - 1}}{\binom{n}{\lfloor cn \rfloor - k}} = \frac{\lfloor cn \rfloor - k}{n - \lfloor cn \rfloor + k + 1}.$$

When k is small compared to n, this ratio is close to $c/(1-c)$ and so, by Principle 11.4, we expect

$$\sum_{k=0}^{\lfloor cn \rfloor} \binom{n}{k} \sim \frac{\binom{n}{\lfloor cn \rfloor}}{1 - (c/(1-c))} = \frac{1-c}{1-2c} \binom{n}{\lfloor cn \rfloor}. \tag{11.37}$$

A table comparing exact and approximate values is given in Figure 11.2. □

We now look at sums with slowly decreasing terms. Such sums can usually be done by interpreting the sum as an approximation to a Riemann sum associated with an integral. We'll merely state the result for the most common case.

c	n	10	20	50	100	200
0.1	A	11.25	213.75	2.384×10^6	1.947×10^{13}	1.815×10^{27}
	E	11	211	2.370×10^6	1.942×10^{13}	1.813×10^{27}
	A/E	1.023	1.01	1.006	1.003	1.002
0.2	A	60	6460	1.370×10^{10}	7.146×10^{20}	2.734×10^{42}
	E	56	6196	1.343×10^{12}	7.073×10^{20}	2.719×10^{42}
	A/E	1.07	1.04	1.02	1.01	1.005
0.4	A	630	377910	1.414×10^{14}	4.124×10^{28}	4.942×10^{57}
	E	386	263950	1.141×10^{14}	3.606×10^{28}	4.568×10^{57}
	A/E	1.6	1.4	1.2	1.14	1.08

Figure 11.2 The exact values (E) and approximate values (A) of the sum in (11.37) for $c = 0.1$, 0.2, 0.4 and $n = 10$, 20, 40, 100, 200. The ratios A/E are also given.

Principle 11.5 Sums of slowly decreasing terms

Let $r_k(n) = t_{n,k+1}/t_{n,k}$ and suppose $\lim_{n\to\infty} r_k(n) = 1$ for all k. Suppose there is a function $f(n) > 0$ with $\lim_{n\to\infty} f(n) = 0$ such that, for all "fairly large" k, $(1 - r_k(n))/k \sim f(n)$ as $n \to \infty$. Then we usually have

$$\sum_{k\geq 0} t_{n,k} \;\sim\; \sqrt{\frac{\pi}{2f(n)}}\, t_{n,0}. \tag{11.38}$$

If the terms in the sum first increase and then decrease and the preceding applies to one half of the sum, then the answer in (11.38) should be doubled.

"Fairly large" means that k is small compared to n but that k is large compared with constants. For example, $(k-1)/kn$ can be replaced by $1/n$.

Example 11.25 Subsets of equal size

Suppose j is constant. How many ways can we choose j distinct subsets of $\underline{n}$ all of the same size? Since there are $\binom{n}{k}$ subsets of size k, you should have no trouble seeing that the answer is

$$\sum_{k=0}^{n} \binom{\binom{n}{k}}{j}.$$

Since the binomial coefficients $\binom{n}{k}$ increase to a maximum at $k = \lfloor n/2 \rfloor$ and then decrease, the same is true for the terms in the sum. Thus we are in Case (c). We can take $K_n = \lfloor n/2 \rfloor$ and we expect (11.38) to apply.

For simplicity, let's treat n as if it is even. The second half of the sum is

$$\sum_{k=0}^{n/2} \binom{\binom{n}{k+n/2}}{j}$$

and so

$$\begin{aligned} r_k \;&=\; \binom{\binom{n}{k+1+n/2}}{j} \Bigg/ \binom{\binom{n}{k+n/2}}{j} \;\approx\; \left(\frac{\binom{n}{k+1+n/2}}{\binom{n}{k+n/2}}\right)^j \\ &=\; \left(\frac{n-k-n/2}{k+1+n/2}\right)^j \;\approx\; \left(\frac{1-2k/n}{1+2k/n}\right)^j \\ &\approx\; e^{-4jk/n} \;\approx\; 1 - 4jk/n, \end{aligned}$$

where the last approximations come from using $1+x \approx e^x$ as $x \to 0$, first with $x = \pm 2k/n$ and then with $x = 4jk/n$. Thus $(1-r_k)/k \approx 4j/n$. So we take $f(n) = 4j/n$ in Principle 11.5. Remembering that we need to double (11.38), we expect that

$$\sum_{k=0}^{n} \binom{\binom{n}{k}}{j} \sim 2\sqrt{\frac{\pi n}{8j}} \binom{\binom{n}{\lfloor n/2 \rfloor}}{j} \sim \sqrt{\frac{\pi n}{2j}} \frac{\binom{n}{\lfloor n/2 \rfloor}^j}{j!}.$$

This is correct. □

Example 11.26 Involutions In Theorem 2.2 (p. 48), we showed that the number of involutions of $\underline{n}$ is

$$I_n = \sum_{k=0}^{\lfloor n/2 \rfloor} \frac{n!}{(n-2k)!2^k k!}.$$

It is not obvious where the maximum term is; however, we can find it by solving the equation $t_{n,k+1}/t_{n,k} = 1$ for k. We have

$$\frac{t_{n,k+1}}{t_{n,k}} = \frac{(n-2k)(n-2k-1)}{2(k+1)} = 1. \tag{11.39}$$

Clearing fractions in the right hand equation leads to a quadratic equation for k whose solution is close to $m = (n-\sqrt{n})/2$. For simplicity, we assume that m is an integer. Since the maximum is not at the end, we expect to apply Principle 11.5. We split the sum into two pieces at m, one of which is

$$\sum_{k=0}^{\sqrt{n}} \frac{n!}{(n-2m-2k)!2^{m+k}(m+k)!}.$$

Adjusting (11.39) for this new index, we have

$$r_k(n) = \frac{t_{n,k+1}}{t_{n,k}} \approx \frac{(\sqrt{n}-2k)^2}{n-\sqrt{n}+2k}.$$

After some approximations and other calculations, we find that $r_k \approx 1-(4k-1)/\sqrt{n}$. Thus $f(n) \sim 4/\sqrt{n}$ and so, doubling (11.38), we expect

$$I_n \sim \frac{2\sqrt{\pi/8}\, n^{1/4}\, n!}{(\sqrt{n})!\, 2^{(n-\sqrt{n})/2}\, \big((n-\sqrt{n})/2\big)!}.$$

If you wish, you can approximate this by using Stirling's formula. After some calculation, we obtain

$$I_n \sim \frac{n^{n/2}}{\sqrt{2}\, e^{n/2-\sqrt{n}+1/4}}. \tag{11.40}$$

This is correct. Here is a comparison of values.

n	5	20	50	100	200
(11.40)	23.6	2.241×10^{10}	2.684×10^{34}	2.340×10^{82}	3.600×10^{192}
I_n	26	2.376×10^{10}	2.789×10^{34}	2.405×10^{82}	3.672×10^{192}
ratio	0.91	0.94	0.96	0.97	0.98

The convergence is not very fast. It can be improved by making a more refined asymptotic estimate, which we will not do. □

Example 11.27 Set partitions We proved in Theorem 11.3 (p. 320) that the number of partitions of an n-set equals

$$B_n \;=\; \frac{1}{e}\sum_{k=0}^{\infty}\frac{k^n}{k!}. \tag{11.41}$$

We'll use Principle 11.5 (p. 346). Taking ratios to find the maximum term:

$$\frac{t_{n,k+1}}{t_{n,k}} \;=\; \frac{(1+1/k)^n}{k+1} \approx \frac{e^{n/k}}{k}.$$

The approximation comes from $1+1/k = \exp(\ln(1+1/k)) \approx e^{1/k}$. We want $e^{n/k}/k = 1$. Taking logarithms, multiplying by k, and rearranging gives $k\ln k = n$. Let s be the solution to $s\ln s = n$. We split the sum at $\lfloor s\rfloor$. It is convenient (and does not affect the answer) to treat s as if it is an integer.

Unfortunately, the next calculations are quite involved. You may want to skip to the end of the paragraph. We now have a sum that starts at s and so, adjusting for shift of index,

$$r_k(n) \;=\; \frac{(1+1/(s+k))^n}{s+k+1} \;\approx\; \frac{e^{n/(s+k)}}{s+k} \;=\; \frac{e^{s\ln s/(s+k)}}{s+k}.$$

Since $\frac{1}{1+x} \approx 1-x$ for small x and since s is large, we have $\frac{s}{s+k} = \frac{1}{1+k/s} \approx 1-k/s$. Using this in our estimate for $r_k(n)$:

$$r_k(n) \;\approx\; \frac{e^{(1-k/s)\ln s}}{s+k} \;=\; \frac{se^{-(k\ln s)/s}}{s+k} \;\approx\; \left(1-\frac{k}{s}\right)\left(1-\frac{k\ln s}{s}\right),$$

where we have used $\frac{1}{1+x} \approx 1-x$ again and also $e^{-x} \approx 1-x$ for small x. When x and y are small, $(1-x)(1-y) \approx 1-(x+y)$. In this case, $x = k/s$ is much smaller than $y = k(\ln s)/s$ so that $x+y \approx y$ and so we finally have

$$r_k(n) \;\approx\; 1-\frac{k\ln s}{s}$$

and so $f(n) = (\ln s)/s$.

Remembering to double and to include the factor of $1/e$ from (11.41), we have (using Stirling's formula on $s!$)

$$B_n \;\sim\; \frac{\sqrt{2\pi s/\ln s}\,s^n}{e\;s!} \;\sim\; \frac{s^{n-s}e^{s-1}}{\sqrt{\ln s}}, \quad \text{where } s\ln s = n \text{ defines } s. \tag{11.42}$$

Here's how (11.42) compares with the exact values.

n	5	20	50	100	200
s	3.76868	9.0703	17.4771	29.5366	50.8939
(11.42)	70.88	6.305×10^{13}	2.170×10^{47}	5.433×10^{115}	7.010×10^{275}
B_n	52	5.172×10^{13}	1.857×10^{47}	4.759×10^{115}	6.247×10^{275}
ratio	1.36	1.22	1.17	1.14	1.22
better	1.03	1.01	1.005	1.003	1.002

The approximation is quite poor. Had we used the factor of $\sqrt{1+\ln s}$ in the denominator of (11.42), the relative error would have been much better, as shown in the last line of the table. How did we obtain the formula with $\sqrt{1+\ln s}$? After obtaining the form with $\sqrt{\ln s}$ and noting how poor the estimate was, we decided to look for a correction by trial and error. Often, a good way to do this is by adjusting in a simple manner the part of the estimate that contains the smallest function of n—in this case, the function $\ln s$. We tried $C+\ln s$ and found that $C=1$ gave quite accurate estimates. □

Generating Functions

In order to study asymptotic estimates from generating functions, it is necessary to know something about singularities of functions. Essentially, a singularity is a point where the function misbehaves in some fashion. The singularities that are encountered in combinatorial problems are nearly all due to either

- attempting to take the logarithm of zero,
- attempting to raise zero to a power which is not a positive integer,

or both. For example, $-\ln(1-x)$ has a singularity at $x=1$. The power of zero requires a bit of explanation. It includes the obviously bad situation of attempting to divide by zero; however, it also includes things like attempting to take the square root of zero. For example, $\sqrt{1-4x}$ has a singularity at $x=1/4$. To explain why a nonintegral power of zero is bad would take us too far afield. Suffice it to say that the fact that A has two square roots everywhere except at $A=0$ is closely related to this problem.

The following is stated as a principle because we need to be more careful about the conditions in order to have a theorem. For combinatorial problems, you can expect the more technical conditions to be satisfied.

Principle 11.6 Nice singularities *Let a_n be a sequence whose terms are positive for all sufficiently large n. Suppose that $A(x)=\sum_n a_n x^n$ converges for some value of $x>0$. Suppose that $A(x)=f(x)g(x)+h(x)$ where*

- *$f(x)=\big(-\ln(1-x/r)\big)^b(1-x/r)^c$, c is not a positive integer and we do not have $b=c=0$;*
- *$A(x)$ does not have a singularity for $-r\le x<r$;*
- *$\lim_{x\to r} g(x)$ exists and is nonzero (call it L);*
- *$h(x)$ does not have a singularity at $x=r$.*

Then it is usually true that

$$a_n \sim \begin{cases} \dfrac{L(\ln n)^b(1/r)^n}{n^{c+1}\Gamma(-c)}, & \text{if } c\neq 0; \\ \dfrac{bL(\ln n)^{b-1}(1/r)^n}{n}, & \text{if } c=0; \end{cases} \tag{11.43}$$

where Γ is the **Gamma function** *which we describe below.*

The value of r can be found by looking for the smallest (positive) singularity of $A(x)$. Often $g(r)$ is defined and then $g(r)=L$. Since, in many cases there is no logarithm term (and so $b=0$), you may find it helpful to rewrite the principle for the special case $b=0$.

The values of the Gamma function $\Gamma(x)$ can be looked up in tables. In practice, the only information you are likely to need about this function is

$$\Gamma(k) = (k-1)! \quad \text{when } k>0 \text{ is an integer}, \quad \Gamma(x+1) = x\Gamma(x) \quad \text{and} \quad \Gamma(1/2) = \sqrt{\pi}.$$

For example, we compute $\Gamma(-1/2)$ by using $\Gamma(1/2) = (-1/2)\Gamma(-1/2)$:

$$\Gamma(-1/2) = -2\Gamma(1/2) = -2\sqrt{\pi}.$$

We begin with a couple of simple examples.

Example 11.28 Derangements Let D_n be the number of permutations of $\underline{n}$ that have no fixed points. We showed in (11.17) that

$$\sum_{n=0}^{\infty} \frac{D_n x^n}{n!} \;=\; \frac{e^{-x}}{1-x}.$$

We apply Principle 11.6 with this as $A(x)$. There is a singularity at $x = 1$ because we are then dividing by zero. Thus $r = 1$ and we have

$$A(x) \;=\; (1-x)^{-1}e^{-x} + 0, \qquad \text{so} \quad f(x) = (1-x)^{-1}, \quad g(x) = e^{-x}, \quad \text{and} \quad h(x) = 0.$$

Thus $b = 0$ and $c = -1$. Since $g(1) = 1/e$, we have $D_n/n! \sim 1/e$. In Example 4.5 (p. 99) we used an explicit formula for D_n to show that D_n is the closest integer to $n!/e$. We get no such error estimate with this approach. □

Example 11.29 Rational generating functions Suppose that $A(x) = \sum_n a_n x^n = p(x)/q(x)$ where $a_n \geq 0$ and $p(x)$ and $q(x)$ are polynomials such that

- r is the smallest zero of $q(x)$; that is, $q(r) = 0$ and $q(s) \neq 0$ if $0 < s < r$;
- the multiplicity of r as a zero is k; that is, $q(x) = s(x)(1-x/r)^k$ where $s(x)$ is a polynomial and $s(r) \neq 0$;
- $p(r) \neq 0$;

We can apply Principle 11.6 with $f(x) = (1-x/r)^{-k}$ and $g(x) = p(x)/s(x)$. Since r is a zero of $q(x)$ of multiplicity k, it follows that this gives an asymptotic formula for a_n:

$$a_n \;\sim\; \frac{p(r)n^{k-1}(1/r)^n}{s(r)\,(k-1)!}. \tag{11.44}$$

We leave it to you to apply this formula to various rational generating functions that have appeared in the text. □

Example 11.30 The average time for Quicksort Let q_n be the average number of comparisons that are needed to Quicksort n long lists. In Example 10.12 (p. 289) we obtained the ordinary generating function

$$Q(x) \;=\; \frac{-2\ln(1-x) - 2x}{(1-x)^2}.$$

We can take $A(x) = Q(x)$, $r = 1$, $f(x) = -\ln(1-x)\,(1-x)^{-2}$ and $g(x) = 2 + \big(2x/\ln(1-x)\big)$. Then $\lim_{x\to 1} g(x) = 2$. From (11.43),

$$q_n \;\sim\; 2\binom{n+1}{n}\ln n \sim 2n\ln n, \tag{11.45}$$

which we also obtained in (10.30) by manipulating an explicit expression for q_n. Here are some numerical results.

n	5	20	50	100	1000
(11.45)	16.09	119.8	391.2	921.0	13816.
q_n	7.4	71.11	258.9	647.9	10986.
ratio	2.2	1.7	1.5	1.4	1.3

The approximation converges very slowly. You might like to experiment with adjusting (11.45) to improve the estimate.

An alternative approach is to write $Q(x)$ as a difference of two functions and deal with each separately: $Q(x) = Q_1(x) - Q_2(x)$ where

$$Q_1(x) \;=\; \frac{-2\ln(1-x)}{(1-x)^2} \qquad \text{and} \qquad Q_2(x) \;=\; \frac{2x}{(1-x)^2}.$$

Now $f_1(x) = -\ln(1-x)(1-x)^{-2}$, $f_2(x) = (1-x)^{-2}$, $g_1(x) = 2$, $g_2(x) = 2x$ and $h_1(x) = h_2(x) = 0$. We obtain $q_{1,n} \sim 2n\ln n$ as before and $q_{2,n} \sim 2n$. Subtracting we again obtain $q_n \sim 2n\ln n$. □

So far we have dealt with generating functions that required relatively little algebra to obtain the asymptotics. We now look at a more complicated situation.

Example 11.31 Binary operations In Example 11.3 (p. 312) we studied z_n, the number of ways a string of n zeroes can be parenthesized to give zero, and obtained the ordinary generating function

$$Z(x) \;=\; \frac{T(x) - 1 + \sqrt{\big(1-T(x)\big)^2 + 4x}}{2},$$

where

$$T(x) \;=\; \frac{1-\sqrt{1-4x}}{2}$$

is the ordinary generating function for all ways to parenthesize the string. (See (11.12) and (11.14).)

To gear up for studying $Z(x)$, we begin with the simple $T(x)$. Of course, $T(x)$ could easily be studied by using the explicit formula for its coefficients, $t_n = \frac{1}{n}\binom{2n-2}{n-1}$; however, the point is to understand how to handle the square root singularity. The square root has a singularity at $r = 1/4$ since it vanishes there. Thus we write

$$T(x) \;=\; f(x)g(x) + h(x), \qquad \text{where} \quad f(x) = (1-x/r)^{1/2}, \quad g(x) = -1/2 \quad \text{and} \quad \text{h(x)=1/2.}$$

From (11.43) we obtain

$$\begin{aligned} t_n \;&=\; a_n \sim (-1/2)\binom{n-3/2}{n}(1/4)^{-n} \\ &\sim\; (-1/2)\Big(n^{-3/2}/\Gamma(-1/2)\Big)4^n \;=\; \frac{4^{n-1}}{\sqrt{\pi}\,n^{3/2}}, \end{aligned} \tag{11.46}$$

since $(-1/2)\Gamma(-1/2) = \Gamma(1/2) = \sqrt{\pi}$.

We're ready for $Z(x)$. Since $r = 1/4$ is a singularity of $T(x)$, it is also a singularity of $Z(x)$. We can have other singularities when the complicates square root is zero; that is, $(1-T(x)^2) + 4x = 0$. We have

$$(1-T(x))^2 + 4x \;=\; \frac{1 + 2\sqrt{1-4x} + (1-4x)}{4} + 4x \;=\; \frac{1 + 6x + \sqrt{1-4x}}{2}.$$

For this to be zero we must have

$$1 + 6x \;=\; -\sqrt{1-4x}. \tag{11.47}$$

Squaring: $1+12x+36x^2 = 1-4x$ and so $16x+36x^2 = 0$. The two solutions are $x = 0$ and $x = -4/9$. Since squaring can introduce false solutions, so we'll have to check them in (11.47). The value $x = 0$ is not a solution to (11.47) and the value $x = -4/9$ is irrelevant since it does not lie in the interval $[-1/4, 1/4]$. Thus $r = 1/4$.

How can we find f, g and h? It seems likely that $f(x) = \sqrt{1-4x}$ since we have $T(x)$ present in the numerator of $Z(x)$. Then $Z(r) = f(r)g(r) + h(r) = h(r)$ since $f(1/4) = 0$. We could simply try to let $h(x)$ be the constant $Z(r)$, which is $(-1+\sqrt{5}\,)/4$. Thus we have

$$f(x) \;=\; \sqrt{1-4x}, \qquad h(x) \;=\; Z(1/4) \;=\; \frac{\sqrt{5}-1}{4}, \qquad g(x) \;=\; \frac{Z(x) - Z(1/4)}{\sqrt{1-4x}}.$$

We need to find L. To simplify expressions, let $s = \sqrt{1-4x}$. We have

$$L \;=\; \lim_{x \to 1/4} \left(\frac{-s + \sqrt{(1+s)^2 + 16x} - \sqrt{5}}{4s} \right). \tag{11.48}$$

n	5	10	20	30	40	50
(11.46)	12.92	4676.	1.734×10^9	9.897×10^{14}	6.740×10^{20}	5.057×10^{26}
t_n	14	4862	1.767×10^9	1.002×10^{15}	6.804×10^{20}	5.096×10^{26}
t ratio	0.92	0.96	0.98	0.987	0.991	0.992
(11.49)	3.571	1293.	4.792×10^8	2.735×10^{14}	1.863×10^{20}	1.398×10^{26}
z_n	5	1381	4.980×10^8	2.806×10^{14}	1.899×10^{20}	1.419×10^{26}
z ratio	0.7	0.94	0.96	0.97	0.98	0.985
z_n/t_n	0.36	0.284	0.282	0.280	0.279	0.279

Figure 11.3 Asymptotic and exact values for t_n and z_n in Example 11.31. The ratios of asymptotic to exact values are given and the ratio z_n/t_n, which we have shown should approach $(5-\sqrt{5})/10 = 0.276393$.

How can we evaluate the limit? Since numerator and denominator approach zero, we can apply l'Hôpital's Rule (a proof can be found in any rigorous calculus text):

Theorem 11.10 l'Hôpital's Rule *Suppose that* $\lim_{x\to a} f(x) = 0$, $\lim_{x\to a} g(x) = 0$, *and* $g(x) \neq 0$ *when* $0 < |x-a| < \delta$ *for some* $\delta > 0$. *Then*

$$\lim_{x\to a} \frac{f(x)}{g(x)} \;=\; \lim_{x\to a} \frac{f'(x)}{g'(x)},$$

provided the latter limit exists.

We leave it to you to apply l'Hôpital's Rule and some algebra to evaluate the limit. We'll use a trick—rewrite everything in terms of s and then use l'Hôpital's Rule:

$$\begin{aligned}
L &= \lim_{s\to 0}\left(\frac{-s+\sqrt{(1+s)^2-4(s^2-1)}-\sqrt{5}}{4s}\right) && \text{by algebra}\\
&= \lim_{s\to 0}\left(\frac{-s+\sqrt{5+2s-3s^2}-\sqrt{5}}{4s}\right) && \text{by algebra}\\
&= \lim_{s\to 0}\left(\frac{-1+(1/2)(5+2s-3s^2)^{-1/2}(2-6s)}{4}\right) && \text{by l'Hôpital's Rule}\\
&= \frac{-1+5^{-1/2}}{4} \;=\; -\frac{5-\sqrt{5}}{20}.
\end{aligned}$$

We finally have

$$z_n \sim -\frac{5-\sqrt{5}}{20}\binom{n-3/2}{n}(1/4)^{-n} \;\sim\; \frac{5-\sqrt{5}}{10}\frac{4^{n-1}}{\sqrt{\pi}\,n^{3/2}}. \qquad 11.49$$

Comparing the result with the asymptotic formula for t_n, the total number of ways to parenthesize the sequence $0 \wedge \ldots \wedge 0$, we see that the fraction of parenthesizations that lead to a value of zero approaches $(5-\sqrt{5})/10$ as $n \to \infty$. Various comparisons are shown in Figure 11.3. □

Example 11.32 Three misfits Although Principle 11.6 applies to many situations of interest in combinatorics, there are also many cases where it fails to apply. Here are three examples.

- No singularities: The EGF for the number of involutions of $\underline{n}$ is $\exp(x + x^2/2)$. This function has no singularities, so our principle fails.
- Zero radius of convergence: The number of simple graphs with vertex set $\underline{n}$ is $g_n = 2^N$ where $N = \binom{n}{2}$. By the Exponential Formula (Theorem 11.5 (p. 321)), the EGF for the number of connected graphs is $\ln(G(x))$, where $G(x)$ is the EGF for g_n. Unfortunately, $G(x)$ only converges at $x = 0$ so our principle fails.
- Bad singularities: Let $p(n)$ be the number of partitions of the integer n. At the end of Example 10.15 (p. 295) we showed that the ordinary generating function for $p(n)$ is

$$P(x) \;=\; \prod_{i=1}^{\infty}(1-x^i)^{-1}.$$

Clearly $r = 1$. Unfortunately, for every real number c, $P(x)/(1-x)^c \to \infty$ as $x \to 1$. Thus our principle does not apply; however, asymptotic information about $p(n)$ can be deduced from the formula for $P(x)$. The methods are beyond this text. (Actually, $P(x)$ behaves even worse than we indicated—it has a singularity for all x on the unit circle.) □

So far we have dealt with generating functions that are given by an explicit formula. This does not always happen. For example, the ordinary generating function for rooted unlabeled full binary trees by number of leaves is given implicitly by

$$T(x) \;=\; \frac{T(x)^2 + T(x^2)}{2} + x.$$

Although any specific t_n can be found, there is no known way to solve for the function $T(x)$. The following principle helps with many such situations. Unfortunately, the validity of its conclusion is a bit shakier than (11.43).

Principle 11.7 Implicit functions *Let a_n be a sequence whose terms are positive for all sufficiently large n. Let $A(x)$ be the ordinary generating function for the a_n's. Suppose that the function $F(x, y)$ is such that $F(x, A(x)) = 0$. If there are positive real numbers r and s such that $F(r, s) = 0$ and $F_y(r, s) = 0$ and if r is the smallest such r, then it is usually true that*

$$a_n \;\sim\; \sqrt{\frac{rF_x(r,s)}{2\pi F_{yy}(r,s)}}\, n^{-3/2} r^{-n}. \tag{11.50}$$

Example 11.33 RP-trees with degree restrictions Let D be a set of nonnegative integers that contains 0. In Exercise 10.4.11 (p. 301), we said an RP-tree was of outdegree D if the number of sons of each vertex lies in D. Let $T_D(x)$ be the ordinary generating function for unlabeled RP-trees of outdegree D by number of vertices. In Exercise 10.4.11, you were asked to show that

$$T_D(x) \;=\; x \sum_{d \in D} T_D(x)^d.$$

It can be shown that $t_D(n) > 0$ for all sufficiently large n if and only if

$$\gcd\{d - 1 : d \in D\} \;=\; 1.$$

We invite you to prove this by looking at the equation for $g(x) = T_D(x)/x$ and using the fact that all sufficiently large multiples of the gcd of a set S can be expressed as a nonnegative linear combination of the elements of S.

Let $F(x,y) = y - x\sum_{d\in D} y^d$. Then r and s in Principle 11.7 are found by solving the equations

$$\begin{aligned} s - r\sum_{d\in D} s^d &= 0, \\ 1 - r\sum_{d\in D} ds^{d-1} &= 0. \end{aligned}$$

With a bit of algebra, we can get an equation in s alone which can be solved numerically and then used in a simple equation to find r:

$$1 = \sum_{\substack{d\in D \\ d>0}} (d-1)s^d, \tag{11.51}$$

$$r = \left(\sum_{d\in D} s^{d-1}\right)^{-1}. \tag{11.52}$$

Finally

$$\frac{rF_x(r,s)}{F_{yy}(r,s)} = \frac{\sum_{d\in D} s^d}{\sum_{d\in D} d(d-1)s^{d-2}}. \tag{11.53}$$

Since the right side of (11.51) is a sum of positive terms, it is a simple matter to solve it for the unique positive solution to any desired accuracy. This result can then be used in (11.52) to find r accurately. Finally, the results can be used in (11.53) and (11.50) to estimate $t_D(n)$. □

Example 11.34 Unlabeled rooted trees In (11.29) we showed that the generating function by vertices for unlabeled rooted trees in which each vertex has outdegree at most three satisfies

$$T(x) = 1 + x\frac{T(x)^3 + 3T(x)T(x^2) + 2T(x^3)}{6}. \tag{11.54}$$

Since $T(x^2)$ and $T(x^3)$ appear here, it does not seem possible to apply Principle 11.7. However, it can be applied—we simply set

$$F(x,y) = 1 - y + x\frac{y^3 + 3T(x^2)y + 2T(x^3)}{6}.$$

The reason why this is permitted is somewhat technical: The singularity r of $T(x)$ turns out to lie in $(0,1)$, which guarantees that $x^3 < x^2 < x$ when x is near r. As a result $T(x^2)$ and $T(x^3)$ are not near a singularity when x is near r.

Even if you did not fully follow the last part of the previous paragraph, you may have noticed the claim that the singularity of $T(x)$ lies in $(0,1)$. We should prove this, but we will not do so. Instead we will be satisfied with noting that our calculations produce a value of $r \in (0,1)$—a circular argument since we needed that fact before beginning.

The equations for r and s are

$$1 - s + r\frac{s^3 + 3T(r^2)s + 2T(r^3)}{6} = 0, \tag{11.55}$$

$$-1 + r\frac{s^2 + T(r^2)}{2} = 0. \tag{11.56}$$

We have a problem here: In order to compute r and s we need to be able to evaluate the function T accurately and we lack an explicit formula for $T(x)$. This can be gotten around as follows.

Suppose we want to know $T(x)$ at some value of $x = p \in (0,1)$. If we knew the values of $T(p^2)$ and $T(p^3)$, we could regard (11.54) as a cubic in the value of $T(p)$ and solve it. On the other hand, if we knew the values of $T((p^2)^2)$ and $T((p^2)^3)$, we could set $x = p^2$ in (11.54) and solve the resulting cubic for $T(p^2)$. On the surface, this does not seem to be getting us anywhere because we keep needing to know $T(p^k)$ for higher and higher powers k. Actually, that is not what we need—we need

to know $T(p^k)$ *with some desired degree of accuracy*. When k is large, p^k is close to zero and so $T(p^k)$ is close to $T(0) = t_0 = 1$. (We remind you that when we derived (11.54) we chose to set $t_0 = 1$.) How large does k need to be for a given accuracy? We won't go into that.

There is another trick that will simplify our work further. Since $s = T(r)$, we can eliminate s from (11.56) and so $r\big(T(r)^2 + T(r^2)\big) = 1$ is the equation for r. We can now use (11.55) to check for errors in our calculations.

Once r and s have been found it is a fairly straightforward matter to apply (11.50). The only issue that may cause some difficulty is the evaluation of

$$F_x(r,s) \;=\; \frac{s^3 + 3T(r^2)s + 2T(r^3)}{6} + r^3T'(r^2)s + r^3T'(r^3)$$

because of the presence of T'. We can differentiate (11.54) with respect to x and solve the result for $T'(x)$ in terms of x, $T'(x^2)$, $T'(x^3)$ and values of T. Using this recursively we can evaluate T' to any desired degree of accuracy, much as we can T. After considerable calculation, we find that

$$t_n \;\sim\; (0.51788\ldots)n^{-3/2}(0.3551817\ldots)^{-n} \qquad 11.57$$

which can be proved. Since we have the results

n	5	10	20	30	40	50
(11.57)	8.2	512.5	5671108.	9.661×10^{10}	1.964×10^{15}	4.398×10^{19}
t_n	8	507	5622109	9.599×10^{10}	1.954×10^{15}	4.380×10^{19}
ratio	1.02	1.01	1.009	1.006	1.005	1.004

the estimate is a good approximation even for small n. □

Exercises

In this set of exercises, "estimate" always means asymptotically; i.e., for large n. Since you will be using the various principles in the text, your results will only be what is probably true; however, the results obtained will, in fact, be correct. Some problems ask you to use alternate methods to obtain asymptotic estimates. We recommend that, after doing such a problem, you reflect on the amount of work each method requires and the accuracy of the result it produced. In many of the exercises, enumeration formulas are given. You need not derive them unless you are asked to do so.

11.4.1. A path enumeration problem leads to the recursion $a_n = 2a_{n-1} + a_{n-2}$ for $n \geq 2$ with initial conditions $a_0 = 1$ and $a_1 = 3$.

(a) Estimate a_n directly from the recursion using Principle 11.1 (p. 341).

(b) Determine the ordinary generating function and use (11.44) to estimate a_n.

(c) Use the ordinary generating function to obtain an explicit formula for a_n and use this formula to estimate a_n.

11.4.2. The recursions

$$\begin{aligned} U_n &= nU_{n-1} + 2U_{n-2} - (n-4)U_{n-3} - U_{n-4} \\ V_n &= (n-1)(V_{n-1} + V_{n-2}) + V_{n-3} \end{aligned}$$

arise in connection with the "menage" problems. Estimate U_n and V_n from the recursions.

11.4.3. In Example 7.13 (p. 211), an upper bound was found for the running time to merge sort a 2^k-long list. Show that the running time to merge sort an n-long list is $\Theta(n \log n)$.

Note: This is for general n, not just $n = 2^k$ and it is for *any* list, not just worst case.

11.4.4. Let a_n be the number of permutations f of $\underline{n}$ such that $f^k(x) = x$ for all $x \in \underline{n}$.

(a) Show that

$$a_n \;=\; \sum_{d|k} \binom{n-1}{d-1} (d-1)!\, a_{n-d},$$

where "$d|k$" beneath the summation sign—read "d divides k"—means that the sum is over all positive integers d such that k/d is an integer.

(b) Estimate a_n from the recursion.

11.4.5. A *functional digraph* is a simple digraph in which each vertex has outdegree 1. (See Exercise 11.2.17.) We say it is connected if the associated graph is connected.

(a) The number of labeled connected n-vertex functional digraphs is

$$\sum_{k=1}^{n} \frac{n!\, n^{n-k+1}}{(n-k)!}.$$

Obtain the estimate $\sqrt{\pi n/2}\, n^{n+1}$ for this summation.

(b) The average number of components in a labeled n-vertex functional digraph is

$$\sum_{k=1}^{n} \frac{n!}{k\, n^k (n-k)!}.$$

Obtain an estimate for this summation.

11.4.6. Let a_n be the number of partitions of $\underline{n}$ into ordered blocks.

(a) Show that $\sum_n a_n x^n/n! = (2 - e^x)^{-1}$

(b) Estimate a_n from the generating function.

*(c) By expanding $(1 - e^x/2)^{-1}$, show that

$$a_n \;=\; \sum_{k=1}^{\infty} \frac{k^n}{2^{k+1}}$$

and use this summation to estimate a_n.

11.4.7. Let S be a set of positive integers and let a_n be the number of permutations f of $\underline{n}$ such that none of the cycles of f have a length in S. Then

$$\sum_{n\ge 0} a_n x^n/n! \;=\; \frac{1}{1-x} \exp\Bigg(-\sum_{k\in S} x^k/k\Bigg).$$

When S is a finite set, estimate a_n.

11.4.8. Let a_n be the number of labeled simple n-vertex graphs all of whose components are cycles.

(a) Show that

$$\sum_{n\ge 0} a_n x^n/n! \;=\; \frac{\exp(-x/2 - x^2/4)}{\sqrt{1-x}}.$$

(b) Obtain an estimate for a_n.

11.4.9. The EGFs for permutations all of whose cycles are odd is $A_o(x) = \sqrt{\frac{1+x}{1-x}}$, and that for permutations all of whose cycles are even is $A_e(x) = (1-x^2)^{-1/2}$.

(a) Why can't our principles for generating functions be used to estimate the coefficients of $A_o(x)$ and $A_e(x)$?

(b) If you apply the relevant principle, it will give the right answer anyway for $A_o(x)$ but not for $A_e(x)$. Apply it.

(c) Show that $a_{e,2n+1} = 0$ and $a_{e,2n} = \binom{2n}{n}(2n)!\, 4^{-n}$ and then use Stirling's formula.

(d) Use $A_o(x) = (1+x)(1-x^2)^{-1/2}$ to obtain formulas for $a_{o,n}$.

11.4.10. Let S be a set of positive integers and let S' be those positive integers not in S. Let $c_n(S)$ be the number of compositions of n all of whose parts lie in S, with $c_0(S) = 1$.

(a) Derive the formula

$$\sum_{n=0}^{\infty} c_n(S)x^n \;=\; \frac{1}{1-\sum_{k\in S} x^k}.$$

(b) Explain how to derive asymptotics for $c_n(S)$ when S is a finite set.

(c) Explain how to derive asymptotics for $c_n(S)$ when S' is a finite set.

11.4.11. A "path" of length n is a sequence $0 = u_0, u_1, \ldots, u_n = 0$ of nonnegative integers such that $u_{k+1} - u_k \in \{-1, 0, 1\}$ for $k < n$. The ordinary generating function for such paths by length is

$$\frac{1}{\sqrt{1-2x-3x^2}}.$$

Estimate the number of such paths by length. (See Exercise 10.3.2.)

11.4.12. In Exercise 10.4.7 (p. 300) we showed that the generating function for an unlabeled binary RP-tree by number of vertices is $(1 - x - \sqrt{1-2x-3x^2})/2x$. Estimate the coefficients.

11.4.13. A certain kind of unlabeled RP-trees have the generating function

$$\frac{1+x^2-\sqrt{(1+x^2)^2-4x}}{2}$$

when counted by vertices. Estimate the number of such trees with n vertices.

11.4.14. Show that the EGF for permutations with exactly k cycles is

$$\frac{1}{k!}\left\{\ln\left(\frac{1}{1-x}\right)\right\}^k$$

and use this to estimate the number of such permutations when k is fixed and n is large.

11.4.15. Let h_n be the number of unlabeled n-vertex RP-trees in which no vertex has outdegree 1 and let $H(x)$ be the ordinary generating function. Show that

$$H(x)^2 - H(x) + \frac{x}{1+x} \;=\; 0$$

and use this to estimate h_n.

11.4.16. Let h_n be the number of rooted labeled n-vertex trees for which no vertex has outdegree 2. It can be shown that the EGF $H(x)$ satisfies $(1+x)H(x) = xe^{H(x)}$. Estimate h_n.

11.4.17. Let D be a set of positive integers. Let a_n be the number of functions f from $\underline{n}$ to the positive integers such that, (a) if f takes on the value k, then it takes on the value i for all $0 < i < k$, and (b) if f takes on the value k the number of times it does so lies in D. It can be shown that the EGF for the a_n's is

$$A(x) \;=\; \left(1 - \sum_{k\in D} x^k/k!\right)^{-1}$$

For whatever D's you can, show how to estimate a_n.

11.4.18. Let the ordinary generating function for the number of rooted unlabeled n-vertex trees in which every vertex has outdegree at most 2 be $T(x)$. For convenience, set $t_0 = 1$. It can be shown that $T(x) = 1 + x\big(T(x)^2 + T(x^2)\big)/2$. Estimate the numbers of such trees.

*11.4.19. The results here relate to Exercise 10.4.1 (p. 298). We suppose that $A(x)$ satisfies Principle 11.6 and that $a_0 = 0$. Our notation will be that of Principle 11.6 and we assume that $b = 0$ for simplicity.

(a) Suppose that $c < 0$. Let $B(x) = A(x)^k$. Show that we expect $b_n/a_n \sim (g(r)n^{-c})^{k-1}\Gamma(-c)/\Gamma(-ck)$.

(b) Suppose that $c > 0$ and $h(r) \neq 0$. Let $B(x) = A(x)^k$. Show that we expect $b_n/a_n \sim kh(r)^{k-1}$.

(c) Suppose that $c = 1/2$ and $A(r) < 1$. Let $B(x) = (1 - A(x))^{-1}$. Derive

$$B(x) \;=\; \frac{1 - h(x) + f(x)g(x)}{(1 - h(x))^2 - (1 - x/r)g(x)^2}$$

and use this to show that we expect $b_n/a_n \sim (1-h(r))^{-2}$. You may assume that the denominator $(1 - h(x))^2 - (1 - x/r)g(x)^2$ does not vanish on the interval $[-r, r]$.

(d) Suppose that $A(x) = 1$ has a solution $s \in (0, r)$. Show that it is unique. Let $B(x) = (1-A(x))^{-1}$. Show that we expect $b_n \sim 1/(A'(s)s^{n+1})$. Prove that the solution $A(s) = 1$ will surely exist if $c < 0$.

(e) Suppose $r < 1$. It can be shown that the radii of convergence of

$$\sum_{k\geq 2} A(x^k)/k \qquad \text{and} \qquad \sum_{k\geq 2}(-1)^{k-1}A(x^k)/k$$

both equal 1. Explain how we could use this fact to obtain asymptotics for sets and multisets in Exercise 10.4.1 (p. 298) using Principle 11.6, if we could handle $e^{A(x)}$ using the principle.

11.4.20. Recall that the generating function for unlabeled full binary RP-trees by number of leaves is

$$\frac{1 - \sqrt{1 - 4x}}{2}.$$

In the following, Exercise 11.4.19 will be useful.

(a) Use Exercise 10.4.1 (p. 298) to deduce information about lists of such trees.

*(b) Use Exercise 10.4.1(c,d) to deduce information about sets and multisets of such trees. *Hint*. Show that

$$\exp\left(-\sqrt{1 - 4x}/2\right) \;=\; \frac{-\sqrt{1 - 4x}}{2}\sum_{k\geq 0}\frac{(1 - 4x)^k}{2^{2k}(2k + 1)!} + \sum_{k\geq 0}\frac{(1 - 4x)^k}{2^k(2k)!}.$$

11.4.21. Let D be a set of nonnegative integers containing 0. Let t_n be the number of rooted labeled n-vertex trees in which the outdegree of every vertex lies in D. Let $T(x)$ the EGF.

(a) Show that $T(x) \;=\; x\sum_{d\in D} T(x)^d/d!$.

*(b) Let $k = \gcd(D)$. Show that $t_n = 0$ when $n + 1$ is not a multiple of k.

(c) For finite D with $\gcd(D) = 1$, show how to estimate t_n.

*11.4.22. For each part of Exercise 11.2.2 except (c), discuss what sort of information about $\mathsf{E}_{\mathcal{T}}$ would be useful for estimating the coefficients of the exponential generating function given there. (See Exercise 11.4.19.)

Notes and References

The text by Sedgewick and Flajolet [15] covers some of the material in this chapter and Chapter 10, and also contains related material.

Further discussion of exponential generating functions can be found in many of the references given at the end of the previous chapter. Other generating functions besides ordinary and exponential ones play a role in mathematics. Dirichlet series play an important role in some aspects of analytic number theory. Apostol [1] gives an introduction to these series; however, some background in number theory or additional reading in his text is required. In combinatorics, almost all generating functions are ordinary or exponential. The next most important class, Gaussian generating functions, is associated with vector spaces over finite fields. Goldman and Rota [7] discuss them.

Lagrange inversion can be regarded as a theorem in complex analysis or as a theorem in combinatorics. In either case, it can be generalized to a set of simultaneous equations in several variables. Garsia and Shapiro [6] prove such a generalization combinatorially and give additional references. For readers familiar with complex analysis, here is a sketch of an analytic proof. Let $\sum a_n x^n = A(x) = g(T(x))$. By the Cauchy Residue Theorem followed by a change of variables,

$$na_n \;=\; \frac{1}{2\pi i}\oint \frac{A'(x)\,dx}{x^n} \;=\; \frac{1}{2\pi i}\oint \frac{g'(T(x))T'(x)\,dx}{x^n} \;=\; \frac{1}{2\pi i}\oint \frac{g'(T)\,dT}{(T/f(T))^n},$$

which equals the coefficient of u^{n-1} in $g'(u)f(u)^n$ by the Cauchy Residue Theorem.

Tutte's work on rooted maps (Exercise 11.2.18) was done in the 1960s. Connections with his work and the (asymptotic) enumeration of polyhedra are discussed in [3].

Pólya's theorem and some generalizations of it were first discovered by Redfield [14] whose paper was overlooked by mathematicians for about forty years. A translation of Pólya's paper together with some notes is available [13]. DeBruijn [4] give an excellent introduction to Pólya's theorem and some of its generalizations and applications. Harary and Palmer [9] discuss numerous applications in graph theory.

Textbooks on combinatorics generally avoid asymptotics. Wilf [16, Ch. 5] has a nice introduction to asymptotics which, in some ways, goes beyond ours. Books, such as the one by Greene and Knuth [8], that deal with analysis of algorithms may have some material on asymptotics. If you are interested in going beyond the material in this text, you should probably look at journal articles. The article by Bender [2] is an introduction to some methods, including ways to deal with the misfits in Example 11.32 (p. 353). (You should note that the hypotheses of Theorem 5 are too weak. A much more extensive discussion has been given by [12]. This has been corrected by Meir and Moon [11].) Our first principle for generating function asymptotics was adapted from the article by Flajolet and Odlyzko [5]. Pólya [13] discusses computing asymptotics for several classes of chemical compounds. A method for dealing with various types of trees, from combinatorial description to asymptotic formula, is discussed by Harary, Robinson and Schwenk [10].

1. Tom M. Apostol, *Introduction to Analytic Number Theory*, 5th ed., Springer-Verlag (1995).
2. Edward A. Bender, Asymptotic methods in enumeration, *SIAM Review* **16** (1974), 485–515. Errata: **18** (1976), 292.
3. Edward A. Bender, The number of three dimensional convex polyhedra. *American Math. Monthly* **94** (1987) 7–21.
4. Nicolas G. deBruijn, Pólya's theory of counting, 144–184 in E.F. Beckenbach (ed.), *Applied Combinatorial Mathematics*, John Wiley (1964).
5. Philippe Flajolet and Andrew Odlyzko, Singularity analysis of generating functions, *SIAM J. Discrete Math.* **3** (1990), 216–240.
6. Adriano M. Garsia and Joni Shapiro, Polynomials of binomial type and Lagrange inversion, *Proc. of the Amer. Math. Soc.* **64** (1977), 179–185.

7. Jay R. Goldman and Gian-Carlo Rota, On the foundations of combinatorial theory. IV. Finite vector spaces and Eulerian generating functions, *Studies in Applied Math.* **49** (1970), 239–258.
8. Daniel H. Greene and Donald E. Knuth, *Mathematics for the Analysis of Algorithms*, 3rd ed., Birkhäuser (1990).
9. Frank Harary and Edgar M. Palmer, *Graphical Enumeration*, Academic Press (1973).
10. Frank Harary, Robert W. Robinson and Allen J. Schwenk, Twenty-step algorithm for determining the asymptotic number of trees of various species, *J. Australian Math. Soc., Series A* **20** (1975), 483–503.
11. Amram Meir and John W. Moon, On an Asymptotic Method in Enumeration, *J. Combinatorial Theory, Series A* **51** (1989), 77–89.
12. Andrew M. Odlyzko, Asymptotic enumeration methods, 1063–1229 in R.L. Graham et al. (eds.), *Handbook of Combinatorics*, vol. 2, Elsevier (1995).
13. George Pólya and Ronald C. Read, *Combinatorial Enumeration of Groups, Graphs, and Chemical Compounds*, Springer-Verlag (1987). A translation with extensive notes of the article: George Pólya, Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen, *Acta Math.* **68** (1937), 145–254.
14. J.H. Redfield, The theory of group-reduced distributions, *American J. Math.* **49** (1927), 433-455.
15. Robert Sedgwick and Philippe Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley (1996).
16. Herbert S. Wilf, *Generatingfunctionology*, 2nd ed., Academic Press (1994).

APPENDIX A

Induction

Let's explore the idea of induction a bit before getting into the formalities.

Example A.1 A simple inductive proof Suppose that $\mathcal{A}(n)$ is an assertion that depends on n. For example, take $\mathcal{A}(n)$ to be the assertion "$n! > 2^n$". In attempting to decide whether or not $\mathcal{A}(n)$ is true, we may first try to check it for some small values of n. In this example, $\mathcal{A}(1) =$ "$1! > 2^1$" is false, $\mathcal{A}(2) =$ "$2! > 2^2$" is false, and $\mathcal{A}(3) =$ "$3! > 2^3$" is false; but $\mathcal{A}(4) =$ "$4! > 2^4$" is true. We could go on like this, checking each value of n in turn, but this becomes quite tiring.

If we're smart, we can notice that to show that $\mathcal{A}(5)$ is true, we can take the true statement $\mathcal{A}(4) =$ "$4! > 2^4$" and multiply it by the inequality $5 > 2$ to get $5! > 2^5$. This proves that $\mathcal{A}(5)$ is true without doing all of the multiplications necessary to verify $\mathcal{A}(5)$ directly. Since $6 > 2$ and $\mathcal{A}(5)$ is true, we can use the same trick to verify that $\mathcal{A}(6)$ is true. Since $7 > 2$, we could use the fact that $\mathcal{A}(6)$ is true to show that $\mathcal{A}(7)$ is true. This, too, becomes tiring.

If we think about what we are doing in a little more generality, we see that if we have verified $\mathcal{A}(n-1)$ for some value of $n > 2$, we can combine this with $n > 2$ to verify $\mathcal{A}(n)$. This is a "generic" or "general" description of how the validity of $\mathcal{A}(n-1)$ can be transformed into the validity of $\mathcal{A}(n)$. Having verified that $\mathcal{A}(4)$ is true and given a valid generic argument for transforming the validity of $\mathcal{A}(n-1)$ into the validity of $\mathcal{A}(n)$, we claim that the statement "$n! > 2^n$ if $n > 3$" has been proved by *induction*. We hope you believe that this is a proof, because the alternative is to list the infinitude of cases, one for each n. □

Here is a formulation of induction:

Theorem A.1 Induction *Let $\mathcal{A}(m)$ be an assertion, the nature of which is dependent on the integer m. Suppose that we have proved $\mathcal{A}(n)$ for $n_0 \le n \le n_1$ and the statement*

> *"If $n > n_1$ and $\mathcal{A}(k)$ is true for all k such that $n_1 \le k < n$, then $\mathcal{A}(n)$ is true."*

Then $\mathcal{A}(m)$ is true for all $m \ge n_0$.

Definition A.1 *The statement "$\mathcal{A}(k)$ is true for all k such that $n_1 \le k < n$" is called the* **induction assumption** *or* **induction hypothesis** *and proving that this implies $\mathcal{A}(n)$ is called the* **inductive step**. *The values of n with $n_0 \le n \le n_1$ are called the* **base cases**.

In many situations, $n_0 = n_1$ and $n_1 = 0$ or 1; however, this is not always true. In fact, our example requires a different value of n_1. Before reading further, can you identify n_1 in Example A.1? Can you identify the induction assumption?

* * * Stop and think about this! * * *

In our example, we started by proving $\mathcal{A}(4) =$ "$4! > 2^4$", so $n_1 = 4$. The induction assumption is:

$$k! > 2^k \text{ is true for all } k \text{ such that } 4 \le k < n.$$

Remark. Sometimes induction is formulated differently. One difference is that people sometimes use $n+1$ in place of n. Thus the statement in the theorem would be

"If $n \ge n_1$ and $\mathcal{A}(k)$ is true for all $n_1 \le k \le n$, then $\mathcal{A}(n+1)$ is true."

Another difference is that some people always formulate it with $n_0 = 1$. Finally, some people restrict the definition of induction even further by allowing you to use only $\mathcal{A}(n)$ to prove $\mathcal{A}(n+1)$ (or, equivalently, only $\mathcal{A}(n-1)$ to prove $\mathcal{A}(n)$), rather than the full range of $\mathcal{A}(k)$ for $n_0 \le k \le n$. Putting these changes together, we obtain the following more restrictive formulation of induction that is found in some texts:

Corollary Restricted induction *Let $\mathcal{A}(m)$ be an assertion, the nature of which is dependent on the integer m. Suppose that we have proved $\mathcal{A}(1)$ and the statement*

"If $n > 1$ and $\mathcal{A}(n-1)$ is true then, $\mathcal{A}(n)$ is true."

Then $\mathcal{A}(m)$ is true for all $m \ge 1$.

Since this is a special case of Theorem A.1, we'll just simply use Theorem A.1. While there is never need to use a simpler form of induction than Theorem A.1, there is sometimes a need for a more general forms of induction. Discussion of these generalizations is beyond the scope of the text.

Example A.2 A summation We would like a formula for the sum of the first n integers. Let us write $S(n) = 1 + 2 + \ldots + n$ for the value of the sum. By a little calculation,

$$S(1) = 1, \quad S(2) = 3, \quad S(3) = 6, \quad S(4) = 10, \quad S(5) = 15, \quad S(6) = 21.$$

What is the general pattern? It turns out that $S(n) = \frac{n(n+1)}{2}$ is correct for $1 \le n \le 6$. Is it true in general? This is a perfect candidate for an induction proof with

$$n_0 = n_1 = 1 \qquad \text{and} \qquad \mathcal{A}(n): \quad \text{"}S(n) = \tfrac{n(n+1)}{2}\text{."} \tag{A.1}$$

Let's prove it. We have shown that $\mathcal{A}(1)$ is true. In this case we need only the restricted induction hypothesis; that is, we will prove the formula for $S(n)$ by using the formula for $S(n-1)$. Here it is (the inductive step):

$$\begin{aligned} S(n) &= 1 + 2 + \cdots + n = \Big(1 + 2 + \cdots + (n-1)\Big) + n \\ &= S(n-1) + n = \frac{(n-1)\big((n-1)+1\big)}{2} + n && \text{by } \mathcal{A}(n-1), \\ &= \frac{n(n+1)}{2} && \text{by algebra.} \end{aligned}$$

This completes the proof. □

Example A.3 Divisibility We will prove that for all integers $x > 1$ and all positive integers n, $x-1$ divides $x^n - 1$. In this case $n_0 = n_1 = 1$, $\mathcal{A}(n)$ is the statement that $x-1$ divides $x^n - 1$. Since $\mathcal{A}(1)$ states that $x-1$ divides $x-1$ it is obvious that $\mathcal{A}(1)$ is true. Now for the induction step. How can we rephrase $\mathcal{A}(n)$ so that it involves $\mathcal{A}(n-1)$? After a bit of thought, you may come up with

$$x^n - 1 \;=\; x(x^{n-1} - 1) + (x - 1). \tag{A.2}$$

Once this is discovered, the rest is easy:

- By $\mathcal{A}(n-1)$, $x^{n-1} - 1$ is divisible by $x - 1$, thus $x(x^{n-1} - 1)$ is divisible by $x - 1$.
- Also, $x - 1$ divides $x - 1$.
- Thus the sum in (A.2) is divisible by $x - 1$.

This completes the proof.

We could have used the same proof as this one for the statement that the polynomial $x - 1$ divides the polynomial $x^n - 1$ for all positive integers n. □

In the last example, the hardest part was figuring out how to use $\mathcal{A}(n-1)$ in the proof of $\mathcal{A}(n)$. This is quite common:

Observation *The difficult part of a proof by induction is often figuring out how to use the inductive hypothesis.*

Simple examples of inductive proofs may not make this clear. Another difficulty that can arise, as happened in Example A.2, may be that we are not even given a theorem but are asked to discover it.

Example A.4 An integral formula We want to prove the formula

$$\int_0^1 x^a (1-x)^b dx = \frac{a!\; b!}{(a+b+1)!}$$

for nonnegative integers a and b.

What should we induct on? We could choose a or b. Let's use b. Here it is with b replaced with n to look more like our usual formulation:

$$\int_0^1 x^a (1-x)^n dx \;=\; \frac{a!\; n!}{(a+n+1)!}. \tag{A.3}$$

This is our $\mathcal{A}(n)$. Also $n_0 = n_1 = 0$, the smallest nonnegative integer. You should verify that $\mathcal{A}(0)$ is true, thereby completing the first step in the inductive proof. (Remember that $0! = 1$.)

How can we use our inductive hypothesis to prove $\mathcal{A}(n)$? This is practically equivalent to asking how to manipulate the integral in (A.3) so that the power of $(1-x)$ is reduced. Finding the answer depends on a knowledge of techniques of integration. Someone who has mastered them will realize that integration by parts could be used to reduce the degree of $1-x$ in the integral. Let's do it. Integration by parts states that $\int u\, dv \;=\; uv - \int v\, du$. We set $u = (1-x)^n$ and $dv = x^a dx$. Then $du = -n(1-x)^{n-1}$, $v = \frac{x^{a+1}}{a+1}$ and

$$\begin{aligned}\int_0^1 x^a (1-x)^n dx \;&=\; \frac{(1-x)^n x^{a+1}}{a+1}\bigg|_0^1 + \frac{n}{a+1}\int_0^1 x^{a+1}(1-x)^{n-1} dx \\ &=\; \frac{n}{a+1}\int_0^1 x^{a+1}(1-x)^{n-1} dx.\end{aligned}$$

The last integral can be evaluated by $\mathcal{A}(n-1)$ and so

$$\int_0^1 x^a (1-x)^n dx = \frac{n}{a+1}\,\frac{(a+1)!\,(n-1)!}{((a+1)+(n-1)+1)!} \;=\; \frac{a!\,n!}{(a+n+1)!}.$$

This completes the inductive step, thereby proving (A.3). □

We now do a more combinatorial inductive proof. This requires a definition from the first chapter. If A is a set, let $|A|$ stand for the number of elements in A.

Definition A.2 Cartesian Product *If $C_1, \ldots, C_k$ are sets, the Cartesian product of the sets is written $C_1 \times \cdots \times C_k$ and consists of all k long lists $(x_1, \ldots, x_k)$ with $x_i \in C_i$ for $1 \le i \le k$.*

Example A.5 Size of the Cartesian product We want to prove

$$|C_1 \times \cdots \times C_n| \;=\; |C_1| \cdots |C_n|. \tag{A.4}$$

This is our $\mathcal{A}(n)$. Since this is trivially true for $n = 1$, it is reasonable to take $n_0 = n_1 = 1$. It turns out that this choice will make the inductive step more difficult. It is better to choose $n_1 = 2$. (Normally one would discover that part way through the proof. To simplify things a bit, we've "cheated" by telling you ahead of time.)

To begin with, we need to prove $\mathcal{A}(2)$. How can this be done? This is the difficult part. Let's postpone it for now and do the inductive step.

We claim that

the sets $C_1 \times \cdots \times C_n$ and $(C_1 \times \cdots \times C_{n-1}) \times C_n$ have the same number of elements.

Why is this? They just differ by pairs of parentheses: Suppose $x_1 \in C_1$, $x_2 \in C_2$, ... and $x_n \in C_n$. Then

$$(x_1, \ldots, x_n) \;\in\; C_1 \times \cdots \times C_n$$

and

$$((x_1, \ldots, x_{n-1}), x_n) \;\in\; (C_1 \times \cdots \times C_{n-1}) \times C_n$$

just differ by pairs of parentheses. Thus

$$\begin{aligned} |C_1 \times \cdots \times C_n| \;&=\; |(C_1 \times \cdots \times C_{n-1}) \times C_n| && \text{by the above,} \\ &=\; |C_1 \times \cdots \times C_{n-1}| \cdot |C_n| && \text{by } \mathcal{A}(2), \\ &=\; \Big(|C_1| \cdots |C_{n-1}|\Big)|C_n| && \text{by } \mathcal{A}(n-1). \end{aligned}$$

This completes the inductive step. Note that this is different from our previous inductive proofs in that we used both $\mathcal{A}(n-1)$ and $\mathcal{A}(2)$ in the inductive step.

We must still prove $\mathcal{A}(2)$. Let $C_1 = \{y_1, \ldots, y_k\}$, where $k = |C_1|$. Then

$$C_1 \times C_2 = (\{y_1\} \times C_2) \cup \cdots \cup (\{y_k\} \times C_2).$$

Since all of the sets in the union are disjoint, the number of elements in the union is the sum of the number of elements in each of the sets separately. Thus

$$|C_1 \times C_2| = \big|\{y_1\} \times C_2\big| + \cdots + \big|\{y_k\} \times C_2\big|.$$

You should be able to see that $|\{y_i\} \times C_2| = |C_2|$. Since the sum has $k = |C_1|$ terms, all of which equal $|C_2|$, we finally have $|C_1 \times C_2| = |C_1| \cdot |C_2|$. □

Our final example requires more than $\mathcal{A}(2)$ and $\mathcal{A}(n-1)$ to prove $\mathcal{A}(n)$.

Example A.6 Every integer is a product of primes A positive integer $n > 1$ is called a *prime* if its only divisors are 1 and n. The first few primes are 2, 3, 5, 7, 11, 13, 17, 19, 23. We now prove that every integer $n > 1$ is a product of primes, where we consider a single prime p to be a product of one prime, itself. We shall prove $\mathcal{A}(n)$:

"Every integer $n \geq 2$ is a product of primes."

We start with $n_0 = n_1 = 2$, which is a prime and hence a product of primes. Assume the induction hypothesis and consider $\mathcal{A}(n)$. If n is a prime, then it is a product of primes (itself). Otherwise, n has a divisor s with $1 < s < n$ and so $n = st$ where $1 < t < n$. By the induction hypotheses $\mathcal{A}(s)$ and $\mathcal{A}(t)$, s and t are each a product of primes, hence $n = st$ is a product of primes. This completes the proof of $\mathcal{A}(n)$ □

In all except Example A.5, we used algebra or calculus manipulations to arrange $\mathcal{A}(n)$ so that we could apply the inductive hypothesis. In Example A.5, we used a set theoretic argument: We found that the elements in the set $C_1 \times \cdots \times C_n$ could be put into one to one correspondence with the elements in the set $(C_1 \times \cdots \times C_{n-1}) \times C_n$. This sort of set-theoretic argument is more common in combinatorial inductive proofs than it is in noncombinatorial ones.

Exercises

In these exercises, indicate clearly

(i) what n_0, n_1 and $\mathcal{A}(n)$ are,

(ii) what the inductive step is and

(iii) where the inductive hypothesis is used.

A.1. Prove that the sum of the first n odd numbers is n^2.

A.2. Prove that $\sum_{k=0}^{n} k^2 = n(n+1)(2n+1)/6$ for $k \geq 0$.

A.3. Conjecture and prove the general formula of which the following are special cases:

$$1 - 4 + 9 - 16 = -(1 + 2 + 3 + 4)$$
$$1 - 4 + 9 - 16 + 25 = 1 + 2 + 3 + 4 + 5.$$

*A.4. Conjecture and prove the general formula of which the following are special cases:

$$\frac{x}{1+x} + \frac{2x^2}{1+x^2} = \frac{x}{1-x} - \frac{4x^4}{1-x^4}$$
$$\frac{x}{1+x} + \frac{2x^2}{1+x^2} + \frac{4x^4}{1+x^4} = \frac{x}{1-x} - \frac{8x^8}{1-x^8}.$$

A.5. Some calculus texts omit the proof of $(x^n)' = nx^{n-1}$ or slide over the inductive nature of the proof. Give a proper inductive proof for $n \geq 1$ using $x' = 1$ and the formula for the derivative of a product.

A.6. Conjecture and prove a formula for $\int_0^\infty x^n e^{-x} dx$ for $n \geq 0$. (Your answer should not include integrals.)

A.7. What is wrong with the following inductive proof that all people have the same sex? Let $\mathcal{A}(n)$ be "In any group of n people, all people are of the same sex." This is obviously true for $n = 1$. For $n > 1$, label the people $P_1, \ldots, P_n$. By the induction assumption, $P_1, \ldots, P_{n-1}$ are all of the same sex and $P_2, \ldots, P_n$ are all of the same sex. Since P_2 belongs to both groups, the sex of the two groups is the same and so we are done.

A.8. We will show by induction that $1 + 2 + \cdots + n = (2n + 1)^2/8$. By the inductive hypothesis, $1 + 2 + \cdots + (n - 1) = (2n - 1)^2/8$. Adding n to both sides and using $n + (2n - 1)^2/8 = (2n + 1)/8$, we obtain the formula. What is wrong with this proof?

A.9. Imagine drawing n distinct straight lines so as to divide the plane into regions in some fashion. Prove that the regions can be assigned numbers 0 and 1 so that if two regions share a line segment in their boundaries, then they are numbered differently.

APPENDIX B

Rates of Growth and Analysis of Algorithms

Suppose we have an algorithm and someone asks us "How good is it?" To answer that question, we need to know what they mean. They might mean "Is it correct?" or "Is it understandable?" or "Is it easy to program?" We won't deal with any of these.

They also might mean "How fast is it?" or "How much space does it need?" These two questions can be studied by similar methods, so we'll just focus on speed. Even now, the question is not precise enough. Does the person mean "How fast is it on this particular problem and this particular machine using this particular code and this particular compiler?" We could answer this simply by running the program! Unfortunately, that doesn't tell us what would happen with other machines or with other problems that the algorithm is designed to handle.

We would like to answer a question such as "How fast is Algorithm 1 for finding a spanning tree?" in such a way that we can compare that answer to "How fast is Algorithm 2 for finding a spanning tree?" and obtain something that is not machine or problem dependent. At first, this may sound like an impossible goal. To some extent it is; however, quite a bit can be said.

How do we achieve machine independence? We think in terms of simple machine operations such as multiplication, fetching from memory and so on. If one algorithm uses fewer of these than another, it should be faster. Those of you familiar with computer instruction timing will object that different basic machine operations take different amounts of time. That's true, but the times are not wildly different. Thus, if one algorithm uses *a lot fewer* operations than another, it should be faster. It should be clear from this that we can be a bit sloppy about what we call an operation; for example, we might call something like $x = a + b$ one operation. On the other hand, we can't be so sloppy that we call $x = a_1 + \cdots + a_n$ one operation if n is something that can be arbitrarily large.

Suppose we have an algorithm for a class of problems. If we program the algorithm in some language and use some compiler to produce code that we run on some machine, then there is a function $f(n)$ that measures the average (or worst, if you prefer) running time for the program on problems

of size n. We want to study how $f(n)$ grows with n, but we want to express our answers in a manner that is independent of the language, compiler, and computer. Mathematicians have introduced notation that is quite useful for studying rates of growth in this manner. We'll study the notation in this appendix.

B.1 The Basic Functions

Example B.1 Let's look at how long it takes to find the maximum of a list of n integers where we know nothing about the order they are in or how big the integers are. Let $a_1, \ldots, a_n$ be the list of integers. Here's our algorithm for finding the maximum.

$max = a_1$
`For` $i = 2, \ldots, n$
    `If` $a_i > max$`, then` $max = a_i$`.`
    `End for`
`Return` max

Being sloppy, we could say that the entire comparison and replacement in the "`If`" takes an operation and so does the stepping of the index i. Since this is done $n-1$ times, we get $2n-2$ operations. There are some setup and return operations, say s, giving a total of $2n-2+s$ operations. Since all this is rather sloppy all we can really say is that for large n and actual code on an actual machine, the procedure will take about Cn "ticks" of the machine's clock. Since we can't determine C by our methods, it will be helpful to have a notation that ignores it. We use $\Theta(f(n))$ to designate any function that behaves like a constant times $f(n)$ for arbitrarily large n. Thus we would say that the "`If`" takes time $\Theta(n)$ and the setup and return takes time $\Theta(1)$. Thus the total time is $\Theta(n) + \Theta(1)$. Since n is much bigger than 1 for large n, the total time is $\Theta(n)$. □

We need to define Θ more precisely and list its most important properties. We will also find it useful to define O, read "big oh."

Definition B.1 Notation for Θ and O *Let f and g be functions from the positive integers to the real numbers.*

- *We say that $g(n)$ is $O(f(n))$ if there exists a positive constant B such that $|g(n)| \leq B|f(n)|$ for all sufficiently large n. In this case we say that g grows no faster than f or, equivalently, that f grows at least as fast as g.*
- *We say that $g(n)$ is $O^+(f(n))$ if $g(n)$ is $O(f(n))$ and $g(n) \geq 0$ for sufficiently large n.*
- *We say that $g(n)$ is $\Theta(f(n))$ if (i) $g(n)$ is $O(f(n))$ and (ii) $f(n)$ is $O(g(n))$. In this case we say that f and g grow at the same rate.*
- *We say that $g(n)$ is $\Theta^+(f(n))$ if $g(n)$ is $\Theta(f(n))$ and $g(n) \geq 0$ for sufficiently large n.*

Remarks **1**. The phrase "$\mathcal{S}(n)$ is true for all sufficiently large n" means that there is some integer N such that $\mathcal{S}(n)$ is true whenever $n \geq N$.

2. Saying that something is $\Theta^+(f(n))$ gives an idea of *how big it is* for large values of n. Saying that something is $O^+(f(n))$ gives an idea of *an upper bound on how big it is* for all large values of n. (We said "idea of" because we don't know what the constants are.)

3. The notation O^+ and Θ^+ is *not standard*. We have introduced it because it is convenient when combining functions.

Theorem B.1 Some properties of Θ and O *We have*

(a) *$g(n)$ is $\Theta(f(n))$ if and only if there are positive constants A and B such that $A|f(n)| \leq |g(n)| \leq B|f(n)|$ for all sufficiently large n.*

(b) *If $g(n)$ is $\Theta^+(f(n))$, then $g(n)$ is $\Theta(f(n))$.*
If $g(n)$ is $O^+(f(n))$, then $g(n)$ is $O(f(n))$.
If $g(n)$ is $\Theta(f(n))$, then $g(n)$ is $O(f(n))$.

(c) *$f(n)$ is $\Theta(f(n))$ and $f(n)$ is $O(f(n))$.*

(d) *If $g(n)$ is $\Theta(f(n))$ and C and D are nonzero constants, then $Cg(n)$ is $\Theta(Df(n))$.*
If $g(n)$ is $O(f(n))$ and C and D are nonzero constants, then $Cg(n)$ is $O(Df(n))$.

(e) *If $g(n)$ is $\Theta(f(n))$, then $f(n)$ is $\Theta(g(n))$.*

(f) *If $g(n)$ is $\Theta(f(n))$ and $f(n)$ is $\Theta(h(n))$, then $g(n)$ is $\Theta(h(n))$.*
If $g(n)$ is $O(f(n))$ and $f(n)$ is $O(h(n))$, then $g(n)$ is $O(h(n))$.

(g) *If $g_1(n)$ is $\Theta(f_1(n))$ and $g_2(n)$ is $\Theta(f_2(n))$, then $g_1(n)g_2(n)$ is $\Theta(f_1(n)f_2(n))$ and, if in addition f_2 and g_2 are never zero, then $g_1(n)/g_2(n)$ is $\Theta(f_1(n)/f_2(n))$.*
If $g_1(n)$ is $O(f_1(n))$ and $g_2(n)$ is $O(f_2(n))$, then $g_1(n)g_2(n)$ is $O(f_1(n)f_2(n))$.

(h) *If $g_1(n)$ is $\Theta^+(f_1(n))$ and $g_2(n)$ is $\Theta^+(f_2(n))$, then $g_1(n)+g_2(n)$ is $\Theta^+\big(\max\big(|f_1(n)|, |f_2(n)|\big)\big)$.*
If $g_1(n)$ is $O(f_1(n))$ and $g_2(n)$ is $O(f_2(n))$, then $g_1(n) + g_2(n)$ is $O\big(\max\big(|f_1(n)|, |f_2(n)|\big)\big)$.

(i) *If $g(n)$ is $\Theta^+(f(n))$ and $h(n)$ is $O^+(f(n))$, then $g(n) + h(n)$ is $\Theta^+(f(n))$.*

Note that by Theorem 5.1 (p. 127) and properties (c), (e) and (f) above, the statement "$g(n)$ is $\Theta(f(n))$" defines an equivalence relation on the set of functions from the positive integers to the reals. Similarly, "$g(n)$ is $\Theta^+(f(n))$" defines an equivalence relation on the set of functions which are positive for sufficiently large n.

Proof: If you completely understand the definitions of O, O^+, Θ, and Θ^+, many parts of the theorem will be obvious to you. None of the parts is difficult to prove and so we leave most of the proofs as an exercise. To help you out, we'll do a couple of proofs.

We'll do (f) for Θ. By (a), there are nonzero constants A_i and B_i such that

$$A_1|f(n)| \le |g(n)| \ \le \ B_1|f(n)|$$

and

$$A_2|h(n)| \le |f(n)| \ \le \ B_2|h(n)|$$

for all sufficiently large n. It follows that

$$A_1A_2|h(n)| \ \le \ A_1|f(n)| \ \le \ |g(n)| \ \le \ B_1|f(n)| \ \le \ B_1B_2|h(n)|$$

for all sufficiently large n. With $A = A_1A_2$ and $B = B_1B_2$, it follows that $g(n)$ is $\Theta(h(n))$.

We'll do (h) for Θ^+. By (a), there are nonzero constants A_i and B_i such that

$$A_1|f_1(n)| \ \le \ g_1(n) \ \le \ B_1|f_1(n)| \quad \text{and} \quad A_2|f_2(n)| \ \le \ g_2(n) \ \le \ B_2|f_2(n)|$$

for sufficiently large n. Adding gives

$$A_1|f_1(n)| + A_2|f_2(n)| \ \le \ g_1(n) + g_2(n) \ \le \ B_1|f_1(n)| + B_2|f_2(n)| \tag{B.1}$$

We now do two things. First, let $A = \min(A_1, A_2)$ and note that

$$A_1|f_1(n)| + A_2|f_2(n)| \ \ge \ A\big(|f_1(n)| + |f_2(n)|\big) \ \ge \ A\max\big(|f_1(n)|, |f_2(n)|\big).$$

Second, let $B = 2\max(B_1, B_2)$ and note that

$$\begin{aligned} B_1|f_1(n)| + B_2|f_2(n)| \ &\le \ \frac{B\big(|f_1(n)| + |f_2(n)|\big)}{2} \\ &\le \ \frac{B\,2\max(|f_1(n)|, |f_2(n)|\big)}{2} \ = \ B\max\big(|f_1(n)|, |f_2(n)|\big). \end{aligned}$$

Using these two results in (B.1) gives

$$A\max\big(|f_1(n)|, |f_2(n)|\big) \ \le \ g_1(n) + g_2(n) \ \le \ B\max(|f_1(n)|, |f_2(n)|\big),$$

which proves that $g_1(n) + g_2(n)$ is $\Theta^+\max\big(|f_1(n)|, |f_2(n)|\big)$. □

Example B.2 Using the notation To illustrate these ideas, we'll consider three algorithms for evaluating a polynomial $p(x)$ of degree n at some point r; i.e., computing $p_0 + p_1r + \cdots + p_nr^n$. We are interested in how fast they are when n is large. Here are the procedures. You should convince yourself that they work.

```
Poly1(n, p, r)
      S = p_0
      For i = 1, ..., n     S = S + p_i*Pow(r, i).
      Return S
End

Pow(r, i)
      P = 1
      For j = 1, ..., i     P = P * r.
      Return P
End
```

```
Poly2(n, p, r)
      S = p_0
      P = 1
      For i = 1, ..., n
            P = P * r
            S = S + p_i * P
      End for
      Return S
End
```

```
Poly3(n, p, r)
      S = p_n
      /* Here i decreases from n to 1. */
      For i = n, ..., 2, 1      S = S * r + p_{i-1}
      Return S
End
```

Let $T_n(\texttt{Name})$ be the time required for the procedure `Name`. Let's analyze `Poly1`. The "`For`" loop in `Pow` is executed i times and so takes Ci operations for some constant $C > 0$. The setup and return in `Pow` takes some constant number of operations $D > 0$. Thus $T_n(\texttt{Pow}) = Ci + D$ operations. As a result, the ith iteration of the "`For`" loop in `Poly1` takes $Ci + E$ operations for some constants C and $E > D$. Adding this over $i = 1, 2, \ldots, n$, we see that the total time spent in the "`For`" loop is $\Theta^+(n^2)$ since $\sum_{i=1}^{n} i = n(n+1)/2$. (You should write out the details.) Since the rest of `Poly1` takes $\Theta^+(1)$ time, $T_n(\texttt{Poly1})$ is $\Theta^+(n^2)$ by Theorem B.1(h).

The amount of time spent in the "`For`" loop of `Poly2` is constant and the loop is executed n times. It follows that $T_n(\texttt{Poly2})$ is $\Theta^+(n)$. The same analysis applies to `Poly3`.

What can we conclude from this about the comparative speed of the algorithms? By Theorem B.1(a), Θ^+, there are positive reals A and B so that $An^2 \le T_n(\texttt{Poly1})$ and $T_n(\texttt{Poly2}) \le Bn$ for sufficiently large n. Thus $T_n(\texttt{Poly2})/T_n(\texttt{Poly1}) \le B/An$. As n gets larger, `Poly2` looks better and better compared to `Poly1`.

Unfortunately, the crudeness of Θ does not allow us to make any distinction between `Poly2` and `Poly3`. What we can say is that $T_n(\texttt{Poly2})$ is $\Theta^+(T_n(\texttt{Poly3}))$; i.e., $T_n(\texttt{Poly2})$ and $T_n(\texttt{Poly3})$ grow at the same rate. A more refined estimate can be obtained by counting the actual number of operations involved. You should compare the number of multiplications required and thereby obtain a more refined estimate. □

So far we've talked about how long an algorithm takes to run as if this were a simple, clear concept. In the next example we'll see that there's an important point that we've ignored.

Example B.3 What is average running time? Let's consider the problem of (a) deciding whether or not a simple graph[1] can be properly colored with four colors and, (b) if a proper coloring exists, producing one. (A proper coloring is a function $\lambda: V \to C$, the set of colors, such that, if $\{u, v\}$ is an edge, then $\lambda(u) \neq \lambda(v)$.) We may as well assume that $V = \underline{n}$ and that the colors are c_1, c_2, c_3 and c_4.

Here's a simple algorithm to determine a λ by using backtracking to go lexicographically through possible colorings $\lambda(1), \lambda(2), \ldots, \lambda(n)$.

1. Initialize: Set $v = 1$ and $\lambda(1) = c_1$.
2. Advance in decision tree: If $v = n$, stop with λ determined; otherwise, set $v = v + 1$ and $\lambda(v) = c_1$.
3. Test: If $\lambda(i) \neq \lambda(v)$ for all $i < v$ for which $\{i, v\} \in E$, go to Step 2.
4. Select next decision: Let j be such that $\lambda(v) = c_j$. If $j < 4$, set $\lambda(v) = c_{j+1}$ and go to Step 3.
5. Backtrack: If $v = 1$, stop with coloring impossible; otherwise, set $v = v - 1$ and go to Step 4.

How fast is this algorithm? Obviously it will depend on the graph. For example, if the subgraph induced by the first five vertices is the complete graph K_5 (i.e., all of the ten possible edges are present), then the algorithm stops after trying to color the first five vertices and discovering that there is no proper coloring. If the last five vertices induce K_5 and the remaining $n - 5$ vertices have no edges, then the algorithm makes $\Theta^+(4^n)$ assignments of the form $\lambda(v) = c_k$.

It's reasonable to talk about the average time the algorithm takes if we expect to give it lots of graphs to look at. Most n vertex graphs will have many sets of five vertices that induce K_5. (We won't prove this.) Thus, we should expect that the algorithm will stop quickly on most graphs. In fact, it can be proved that the average number of assignments of the form $\lambda(v) = c_k$ that are made is $\Theta^+(1)$. This means that the average running time of the algorithm is bounded for all n, which is quite good!

Now suppose you give this algorithm to a friend, telling him that on average the running time is practically independent of the number of vertices. He thanks you profusely for such a wonderful algorithm and puts it to work coloring randomly generated "planar" graphs. By the Four Color Theorem, every planar graph can be properly colored with four colors, so the algorithm must make at least n assignments of the form $\lambda(v) = c_k$. (Actually it will almost surely make *many, many* more.) Your friend soon comes back to you complaining bitterly.

What went wrong? In our previous discussion we were averaging over all simple graphs with n vertices. Your friend was interested in the average over all simple planar graphs with n vertices. These averages are *very* different! There is a moral here:

You must be VERY clear what you are averaging over.

Because situations like this do occur in real life, computer scientists are careful to specify what kind of running time they are talking about; either the average of the running time over some reasonable, clearly specified set of problems or the worst (longest) running time over all possibilities. □

[1] A "simple graph" is a set V, called vertices, and a set E of two element subsets of V, called edges. One thinks of an edge $\{u, v\}$ as joining the two vertices u and v.

Example B.4 Which algorithm is faster? Suppose we have two algorithms for a problem, one of which is $\Theta^+(n^2)$ and the other of which is $\Theta^+(n^2 \ln\ln n)$. Which is better?[2] It would seem that the first algorithm is better since n^2 grows slower than $n^2 \ln\ln n$. That's true if n is large enough. How large is large enough? In other words, what is the *crossover point*, the time when we should switch from one algorithm to the other in the interests of speed? To decide, we need to know more than just $\Theta^+(\ )$ because that notation omits constants. Suppose one algorithm has running time close to $3n^2$, and the other, close to $n^2 \ln\ln n$. The second algorithm is better as long as $3 > \ln\ln n$. Exponentiating twice, we get $n < e^{e^3}$, which is about 5×10^8. This is a large crossover point! On the other hand, suppose the first algorithm's running time is close to n^2. In that case, the second algorithm is better as long as $1 > \ln\ln n$, that is, $n < e^e$, which is about 15. A slight change in the constant caused a huge change in the crossover point!

If slight changes in constants matter this much in locating crossover points, what good is $\Theta^+(\ )$ notation? We've misled you! The crossover points are not that important. What matters is how much faster one algorithm is than another. If one algorithm has running time An^2 and the other has running time $Bn^2 \ln\ln n$, the ratio of their speeds is $(B/A) \ln\ln n$. This is fairly close to B/A for a large range of n because the function $\ln\ln n$ grows so slowly. In other words, the running time of the two algorithms differs by nearly a constant factor for practical values of n.

We're still faced with a problem when deciding between two algorithms since we don't know the constants. Suppose both algorithms are $\Theta^+(n^2)$. Which do we choose? If you want to be *certain* you have the faster algorithm, you'll either have to do some very careful analysis of the code or run the algorithms and time them. However, there is a rule of thumb that works pretty well: More complicated data structures lead to larger constants.

Let's summarize what we've learned in the last two paragraphs. Suppose we want to choose the faster of Algorithm A whose running time is $\Theta^+(a(n))$ and Algorithm B whose running time is $\Theta^+(b(n))$.)

- If possible, simplify $a(n)$ and $b(n)$ and ignore all slowly growing functions of n such as $\ln\ln n$. ("What about $\ln n$?" you ask. That's a borderline situation. It's usually better to keep it.) This gives two new functions $a^*(n)$ and $b^*(n)$.
- If $a^*(n) = \Theta^+(b^*(n))$, choose the algorithm with the simpler data structures; otherwise, choose the algorithm with the smaller function.

These rules are far from foolproof, but they provide some guidance. □

We now define two more notations that are useful in discussing rate of growth. The notation o is read "little oh". There is no standard convention for reading $\sim$.

Definition B.2 Notation for o and $\sim$ *Let f, g and h be functions from the positive integers to the real numbers.*

- *If $\lim_{n\to\infty} g(n)/f(n) = 1$, we say that $g(n) \sim f(n)$. In this case, we say that f and g are* **asymptotically equal**.
- *If $\lim_{n\to\infty} h(n)/f(n) = 0$, we say that $hf(n)$ is $o(g(n))$. In this case, we say that h grows slower than f or, equivalently, that f grows faster than h.*

[2] This situation actually occurs, see the discussion at the end of Example 6.3 (p. 152).

Asymptotic equality has many of the properties of equality. The two main exceptions are cancellation and exponentiation:

- You should verify that $n^2 + 1 \sim n^2 + n$ and cancelling the n^2 from both sides is bad because $1 \sim n$ is *false*.
- You should verify that exponentiation is bad by showing that $e^{n^2+1} \sim e^{n^2+n}$ is *false*. In fact, you should verify that e^{n^2+1} is $o(e^{n^2+n})$.

In the following theorem, we see that asymptotic equality defines an equivalence relation (d), allows multiplication and division (c, e), and allows addition of functions with the same sign (f). You should verify that Theorem B.1 says the same thing for $\Theta(\ )$.

Theorem B.2 Some properties of o and $\sim$ *We have*

(a) *If $g(n)$ is $o(f(n))$, then $g(n)$ is $O(f(n))$.*
If $f(n) \sim g(n)$, then $f(n)$ is $\Theta(g(n))$.

(b) *$f(n)$ is* **not** *$o(f(n))$.*

(c) *If $g(n)$ is $o(f(n))$ and C and D are nonzero constants, then $Cg(n)$ is $o(Df(n))$.*
If $g(n) \sim f(n)$ and C is a nonzero constant, then $Cg(n) \sim Cf(n)$.

(d) *"$g(n) \sim f(n)$" defines and equivalence relation.*

(e) *If $g_1(n)$ is $o(f_1(n))$ and $g_2(n)$ is $o(f_2(n))$, then $g_1(n)g_2(n)$ is $o(f_1(n)f_2(n))$.*
If $g_1(n) \sim f_1(n)$ and $g_2(n) \sim f_2(n)$, then $g_1(n)g_2(n) \sim f_1(n)f_2(n)$
and $g_1(n)/g_2(n) \sim f_1(n)/f_2(n)$.

(f) *If $g_1(n)$ is $o(f_1(n))$ and $g_2(n)$ is $o(f_2(n))$, then $g_1(n) + g_2(n)$ is $o(\max(f_1(n), f_2(n)))$.*
If $g_1(n) \sim f_1(n)$ and $g_2(n) \sim f_2(n)$ and $f_1(n)$ and $g_1(n)$ are nonnegative for all sufficiently large n, then $g_1(n) + g_2(n) \sim f_1(n) + f_2(n)$.

(g) *If $h(n)$ is $o(f(n))$ and $g(n) \sim f(n)$, then $g(n) + h(n) \sim f(n)$.*
If $h(n)$ is $o(f(n))$ and $g(n)$ is $\Theta(f(n))$, then $g(n) + h(n)$ is $\Theta(f(n))$.
If $h(n)$ is $o(f(n))$ and $g(n)$ is $O(f(n))$, then $g(n) + h(n)$ is $O(f(n))$.

(h) *If $g_1(n)$ is $o(f_1(n))$ and $g_2(n)$ is $O(f_2(n))$, then $g_1(n)g_2(n)$ is $o(f_1(n)f_2(n))$.*

The proof is left as an exercise. We'll see some applications in the next section.

Exercises

B.1.1. Prove the properties of $\Theta(\ )$, $\Theta^+(\ )$, $O(\ )$, and $O^+(\ )$ given in Theorem B.1.

B.1.2. Prove by example that (g) in Theorem B.1 does not hold for Θ.

B.1.3. Prove or disprove the statement:
"$g(n)$ is $O(f(n))$" defines an equivalence relation for functions from the positive integers to the nonnegative reals (as did the corresponding statement for Θ).

B.1.4. In each case, prove that $f(x)$ is $\Theta(g(x))$ using the definition of $\Theta(\ )$.

(a) $f(x) = x^3 + 5x^2 + 10$, $g(x) = 20x^3$.

(b) $f(x) = x^2 + 5x^2 + 10$, $g(x) = 200x^2$.

B.1.5. In each case, show that the given series has the indicated property.

(a) $\sum_{i=1}^{n} i^2$ is $\Theta(n^3)$.

(b) $\sum_{i=1}^{n} i^3$ is $\Theta(n^4)$.

(c) $\sum_{i=1}^{n} i^{1/2}$ is $\Theta(n^{3/2})$.

B.1.6. Show each of the following

(a) $\sum_{i=1}^{n} i^{-1}$ is $\Theta(\log_b(n))$ for any base $b > 1$.

(b) $\log_b(n!)$ is $O(n \log_b(n))$ for any base $b > 1$

B.1.7. We have three algorithms for solving a problem for graphs. Suppose algorithm A takes n^2 milliseconds to run on a graph with n vertices, algorithm B takes $100n$ milliseconds and algorithm C takes $100(2^{n/10} - 1)$ milliseconds.

(a) Compute the running times for the three algorithms with $n = 5$, 10, 30, 100 and 300. Which algorithm is fastest in each case? slowest?

(b) Which algorithm is fastest for all very large values of n? Which is slowest?

B.1.8. Prove Theorem B.2.

B.1.9. Let $p(x)$ be a polynomial of degree k with positive leading coefficient and suppose that $a > 1$. Prove the following.

(a) $\Theta(p(n))$ is $\Theta(n^k)$.

(b) $O(p(n))$ is $O(n^k)$.

(c) $p(n) = o(a^n)$. (Also, what does this say about the speed of a polynomial time algorithm versus one which takes exponential time?)

(d) $O(a^{p(n)})$ is $O(a^{Cn^k})$ for some $C > 0$.

(e) Unless $p(x) = p_1 x^k + p_2$ for some p_1 and p_2, there is no C such that $a^{p(n)}$ is $\Theta(a^{Cn^k})$.

B.1.10. Suppose $1 < a < b$ and $f(n) \to +\infty$ as $n \to \infty$. Prove that

$$a^{f(n)} = o(b^{f(n)}) \qquad \text{and} \qquad a^{g(n)} = o(a^{f(n)+g(n)}),$$

for all functions $g(n)$.

B.1.11. Consider the following algorithm for determining if the distinct integers $a_1, a_2, \ldots, a_n$ are in increasing order.

```
For i = 2, ..., n
     If a_{i-1} > a_i, return ‘‘out of order.’’
End for
Return ‘‘in order.’’
```

(a) Discuss worst case running time.

(b) Discuss average running time for all permutations of $\underline{n}$.

(c) Discuss average running time for all permutations of $\underline{2n}$ such that $a_1 < a_2 < \cdots < a_n$.

B.2 Doing Arithmetic

If we try to use Theorems B.1 and B.2 in a series of calculations, the lack of arithmetic notation becomes awkward. You're already familiar with the usefulness of notation; for example, when one understands the notation, it is easier to understand and verify the statement

$$(ax - b)^2 = a^2x^2 - 2abx + b^2$$

than it is to understand and verify the statement

> The square of the difference between the product of a and x and b equals the square of a times the square of x decreased by twice the product of a, b and x and increased by the square of b.

If we simply interpret "is" in the theorems as an equality, we run into problems. For example, we would then say that since $n = O(n)$ and $n + 1 = O(n)$, we would have $n = n + 1$. How can we introduce arithmetic notation and avoid such problems? The key is to redefine Θ, O and o slightly using sets. Let Θ^*, O^* and o^* be our old definitions. Our new ones are:

- $\Theta(f(n)) = \{g(n) \mid g(n) \text{ is } \Theta^*(f(n))\}$,
- $\Theta^+(f(n)) = \{g(n) \mid g(n) \text{ is } \Theta^*(f(n)) \text{ and } g(n) \text{ is positive for large } n\}$,
- $O(f(n)) = \{g(n) \mid g(n) \text{ is } O^*(f(n))\}$,
- $O^+(f(n)) = \{g(n) \mid g(n) \text{ is } O^*(f(n)) \text{ and } g(n) \text{ is positive for large } n\}$,
- $o(f(n)) = \{g(n) \mid g(n) \text{ is } o^*(f(n))\}$.

If we replace "is" with "is in", Theorems B.1 and B.2 are still correct. For example, the last part Theorem B.1(b) becomes

> If $g(n) \in \Theta(f(n))$, then $g(n) \in O(f(n))$.

We want to make two other changes:

- Replace functions, numbers, and so on with sets so that we use $\subseteq$ instead of $\in$. For example, instead of $5n \in O(n)$, we say $\{5n\} \subseteq O(n)$.
- An arithmetic operation between sets is done element by element; for example,

$$A + B = \{a + b \mid a \in A \text{ and } b \in B\}.$$

Let's rewrite parts of Theorem B.1 using this notation:

(b) If $\{g(n)\} \subseteq \Theta^+(f(n))$, then $\{g(n)\} \subseteq \Theta(f(n))$.
If $\{g(n)\} \subseteq O^+(f(n))$, then $\{g(n)\} \subseteq O(f(n))$.
If $\{g(n)\} \subseteq \Theta(f(n))$, then $\{g(n)\} \subseteq O(f(n))$.

(f) If $\{g(n)\} \subseteq \Theta(f(n))$ and $\{f(n)\} \subseteq \Theta(h(n))$, then $\{g(n)\} \subseteq \Theta(h(n))$.
If $\{g(n)\} \subseteq O(f(n))$ and $\{f(n)\} \subseteq O(h(n))$, then $\{g(n)\} \subseteq O(h(n))$.

(i) If $\{g(n)\} \subseteq \Theta^+(f(n))$ and $\{h(n)\} \subseteq O^+(f(n))$, then $\{g(n) + h(n)\} \subseteq \Theta^+(f(n))$.

We leave it to you to translate other parts and to translate Theorem B.2.

In practice, people simplify the notation we've introduced by replacing things like $\{f(n)\}$ with $f(n)$, which is good since it makes these easier to read. They also replace $\subseteq$ with $=$, which is dangerous but is, unfortunately, the standard convention. We'll abide by these conventions, but will remind you of what we're doing by footnotes in the text.

Example B.5 Using the notation The statement $f(n) \sim g(n)$ is equivalent to the statement $f(n) = g(n)(1 + o(1))$ and also to $f(n) = g(n) + o(g(n))$. The first is because $f(n)/g(n) \to 1$ if and only if $f(n)/g(n) = 1 + o(1)$. The second follows from

$$g(n)(1+o(1)) = g(n)+g(n)o(1) = g(n)+o(g(n)) \quad \text{and} \quad g(n)+o(g(n)) = g(n)+g(n)o(1) = g(n)(1+o(1)).$$

Why do we need the second of these statements?

* * * Stop and think about this! * * *

Remember that $=$ really means $\subseteq$, so the first statement shows that $g(n)(1 + o(1)) \subseteq g(n) + o(g(n))$ and the second shows that $g(n) = o(g(n)) \subseteq g(n)(1 + o(1))$. Taken together, the two statements show that the sets $g(n)(1 + o(1))$ and $g(n) + o(g(n))$ are equal and so $f(n)$ is in one if and only if it is in the other.

We can include functions of sets: Suppose S is a subset of the domain of the function F, define $F(S) = \{f(s) \mid s \in S\}$. With this notation,

$$e^{o(1)} \;=\; 1 + o(1) \;=\; e^{o(1)} \qquad \text{and} \qquad e^{O(1)} \;=\; O^+(1);$$

however, $O^+(1) \neq e^{O(1)}$. Why is this?

* * * Stop and think about this! * * *

We have $e^{-n} \in O^+(1)$ but, since $n \notin O(1)$, $e^{-n} \notin e^{O(1)}$ □

Everything we've done so far is with functions from the positive integers to the reals and we've asked what happens as $n \to \infty$. We can have functions on other sets and ask what happens when we take a different limit. For example, the definition of a derivative can be written as

$$\frac{f(x+h) - f(x)}{h} \;\sim\; f'(x) \quad \text{as } h \to 0,$$

provided $f'(x) \neq 0$. Taylor's theorem with remainder can be written

$$f(x) \;=\; \sum_{k=0}^{n} \frac{f^{k)}(0)x^k}{k!} + O(x^{n+1}) \quad \text{as } x \to 0,$$

provided $f^{(n+1)}(x)$ is well behaved near $x = 0$. Of course, this is not as good as the form in your calculus book because it says nothing about how big the error term, $O(x^{n+1})$ is for a particular function $f(x)$.

B.3 NP-Complete Problems

Computer scientists talk about "polynomial time algorithms." What does this mean? Suppose that the algorithm can handle arbitrarily large problems and that it takes $\Theta(n)$ seconds on a problem of "size" n. Then we call it a linear time algorithm. More generally, if there is a (possibly quite large) integer k such that the worst case running time on a problem of "size" n is $O(n^k)$, then we say the algorithm is polynomial time.

You may have noticed the quotes around size and wondered why. It is necessary to specify what we mean by the size of a problem. Size is often interpreted as the number of bits required to specify the problem in binary form. You may object that this is imprecise since a problem can

be specified in many ways. This is true; however, the number of bits in one "reasonable" representation doesn't differ too much from the number of bits in another. We won't pursue this further.

If the worst case time for an algorithm is polynomial, theoretical computer scientists think of this as a good algorithm. (This is because polynomials grow relatively slowly; for example, exponential functions grow much faster.) The problem that the algorithm solves is called *tractable*.

Do there exist *intractable problems*; i.e., problems for which no polynomial time algorithm can ever be found? Yes, but we won't study them here. More interesting is the fact that there are a large number of practical problems for which

- no polynomial time algorithm is known and
- no one has been able prove that the problems are intractable.

We'll discuss this a bit. Consider the following problems.

- Coloring Problem: For any $c > 2$, devise an algorithm whose input can be any simple graph and whose output answers the question "Can the graph be properly colored in c colors?"
- Traveling Salesman Problem: For any B, devise an algorithm whose input can be any $n > 0$ and any edge labeling $\lambda: \mathcal{P}_2(\underline{n}) \to \mathbb{R}$ for K_n, the complete graph on n vertices. The algorithm must answer the question "Is there a cycle through all n vertices with cost B or less?" (The cost of a cycle is the sum of $\lambda(e)$ over all e in the cycle.)
- Language Recognition Problem: Devise an algorithm whose input is two finite sets S and T and an integer k. The elements of S and T are finite strings of zeroes and ones. The algorithm must answer the question "Does there exist a finite automaton with k states that accepts all strings in S and accepts none of the strings in T?"

No one knows if these problems are tractable, but it is known that, if one is tractable, then they all are. There are hundreds more problems that people are interested in which belong to this particular list in which all or none are tractable. These problems are called *NP-complete* Many people regard deciding if the NP-complete problems are tractable to be one of the foremost open problems in theoretical computer science.

The NP-complete problems have an interesting property which we now discuss. If the algorithm says "yes," then there must be a specific example that shows why this is so (an assignment of colors, a cycle, an automaton). There is no requirement that the algorithm actually produce such an example. Suppose we somehow obtain a coloring, a cycle or an automaton which is claimed to be such an example. Part of the definition of NP-complete requires that we be able to check the claim in polynomial time. Thus we can check a purported example quickly but, so far as is known, it may take a long time to determine if such an example exists. In other words, I can check your guesses quickly but I don't know how to tell you quickly if any examples exist.

There are problems like the NP-complete problems where no one knows how to do any checking in polynomial time. For example, modify the traveling salesman problem to ask for the minimum cost cycle. No one knows how to verify in polynomial time that a given cycle is actually the minimum cost cycle. If the modified traveling salesman problem is tractable, so is the one we presented above: You need only find the minimum cost cycle and compare its cost to B. Such problems are called *NP-hard* because they are at least as hard as NP-complete problems. A problem which is tractable if the NP-complete problems are tractable is called *NP-easy*.

Some problems are both NP-easy and NP-hard but may not be NP-complete. Why is this? NP-complete problems must ask a "yes/no" type of question and it must be possible to check a specific example in polynomial time as noted in the previous paragraph.

What can we do if we cannot find a good algorithm for a problem? There are three main types of partial algorithms:

1. Almost good: It is polynomial time for all but a very small subset of possible problems. (If we are interested in all graphs, our coloring algorithm in Example B.3 is almost good for any fixed c.)
2. Almost correct: It is polynomial time but in some rare cases does not find the correct answer. (If we are interested in all graphs and a fixed c, automatically reporting that a large graph can't be colored with c colors is almost correct—but it is rather useless.) In some situations, a fast almost correct algorithm can be useful.
3. Close: It is a polynomial time algorithm for a minimization problem and comes close to the true minimum. (There are useful close algorithms for approximating the minimum cycle in the Traveling Salesman Problem.)

Some of the algorithms make use of random number generators in interesting ways. Unfortunately, further discussion of these problems is beyond the scope of this text.

Exercises

B.3.1. The *chromatic number* $\chi(G)$ of a graph G is the least number of colors needed to properly color G. Using the fact that the problem of deciding whether a graph can be properly colored with c colors is NP-complete, prove the following.

(a) The problem of determining $\chi(G)$ is NP-hard.

(b) The problem of determining $\chi(G)$ is NP-easy.
Hint. You can color G with c colors if and only if $c \geq \chi(G)$.

B.3.2. The *bin packing problem* can be described as follows. Given a set S of positive integers and integers B and K, is there a partition of S into K blocks so that the sum of the integers in each block does not exceed B? This problem is known to be NP-complete.

(a) Prove that the following modified problem is NP-easy and NP-hard. Given a set S of positive integers and an integer B, what is the smallest K such that the answer to the bin packing problem is "yes?"

(b) Call the solution to the modified bin packing problem $K(S, B)$. Prove that

$$K(S, B) \ \geq \ \frac{1}{B} \sum_{s \in S} s.$$

(c) The "First Fit" algorithm obtains an upper bound on $K(S, B)$. We now describe it. Start with an infinite sequence of boxes (bins) $B_1, B_2, \ldots$. Each box can hold any number of integers as long as their sum doesn't exceed K. Let $s_1, s_2, \ldots$ be some ordering of S. If the s_i's are placed in the B_j's, the nonempty boxes form an ordered partition of S and so the number of them is an upper bound for $K(S, B)$. For $i = 1, 2, \ldots, |S|$, place s_i in the B_j with the lowest index such that it will not make the sum of the integers in B_j exceed K. Estimate the running time of the algorithm in terms of $|S|$, B and the number of bins actually used.

(d) Call the bound on K obtained by the First Fit algorithm $FF(S, B)$. Prove that $FF(S, B) < 2K(S, B) + 1$.
Hint. When First Fit is done, which bins can be at most half full?

Notes and References

Many texts discuss the notation for rate of growth. A particularly nice introduction is given in Chapter 9 of Graham, Knuth, and Patashnik [4].

Entire textbooks are devoted primarily to the analysis of algorithms. Examples include the books by Aho, Hopcroft and Ullman [1], Baase [2], Knuth [6], Manber [7], Papadimitiriou and Steiglitz [8], and Wilf [9]. There is an extensive journal literature on NP-completeness. The classic book by Garey and Johnson [3] discusses many of the examples. Other discussions can be found in the texts by Papadimitriou and Steiglitz [8] and by Wilf [9] and in the article by Hartmanis [5].

1. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).
2. Sara Baase, *Computer Algorithms: Introduction to Design and Analysis*, 3rd ed., Addison-Wesley (1999).
3. Michael R. Garey and David S. Johnson, *Computers and Intractability. A Guide to the Theory of NP-Completeness*, W.H. Freeman (1979).
4. Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, *Concrete Mathematics*, 2nd ed., Addison-Wesley, Reading (1994).
5. Juris Hartmanis, *Overview of computational complexity theory*, Proceedings of Symposia in Applied Mathematics *38* (1989), 1–17.
6. Donald E. Knuth, *The Art of Computer Programming. Vol. 1: Fundamental Algorithms,* 3rd ed.; *Vol. 2: Seminumerical Algorithms,* 3nd ed.; *Vol. 3: Sorting and Searching,* 3nd ed.; Addison-Wesley (1997, 1997, 1998).
7. Udi Manber, *Introduction to Algorithms: A Creative Approach*, Addison-Wesley (1989).
8. Christos H. Papadimitriou and Kenneth Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Dover (1998).
9. Herbert S. Wilf, *Algorithms and Complexity*, Prentice-Hall (1986).

APPENDIX C

Basic Probability

This appendix is a rapid introduction to the concepts from probability theory that are needed in the text. It is not intended to substitute for a basic course in probability theory, which we strongly recommend for anyone planning either to apply combinatorics (especially in computer science) or to delve into combinatorics for its own sake.

C.1 Probability Spaces and Random Variables

For simplicity, we will limit our attention to finite probability spaces and to real-valued random variables.

Definition C.1 Finite probability space *A* **finite probability space** *is a finite set S together with a function $\Pr : S \to \mathbb{R}$ such that*

$$0 \leq \Pr(s) \leq 1 \quad \text{for all } s \in S \qquad \text{and} \qquad \sum_{s \in S} \Pr(s) = 1.$$

We call the elements of S **elementary events** *and the subsets of S* **events***. For $T \subseteq S$, we define*

$$\Pr(T) = \sum_{t \in T} \Pr(t).$$

Note that $\Pr(\emptyset) = 0$, $\Pr(S) = 1$ and $\Pr(s) = \Pr(\{s\})$ for $s \in S$.

If $\mathcal{A}(s)$ is a statement that makes sense for $s \in S$, we define $\Pr(\mathcal{A}) = \Pr(T)$, were T is the set of all $t \in S$ for which $\mathcal{A}(t)$ is true.

One often has $\Pr(s) = 1/|S|$ for all $s \in S$. In this case, $\Pr(T) = |T|/|S|$, the fraction of elements in S that lie in T. In this case, we call $\Pr$ the **uniform distribution** *on S.*

The terminology "event" can sometimes be misleading. For example, if S consists of all $2^{|A|}$ subsets of $|A|$, an elementary event is a subset of A. Suppose Pr is the uniform distribution on S. If T is the event consisting of all subsets of size k, then $\Pr(T)$ is the fraction of subsets of size k. We say that $\Pr(T)$ is the probability that a subset of A chosen *uniformly at random* has size k. "Uniformly" is often omitted and we simply say that the probability a randomly chosen subset has size k is $|T|/|S| = \binom{|A|}{k}2^{-|A|}$. In statement notation,

$$\Pr(\text{a subset has size } k) \;=\; \binom{|A|}{k}2^{-|A|}.$$

The notion of the probability of a statement being true is neither more nor less general than the notion of the probability of a subset of the probability space S. To see this, note that

- with any statement $\mathcal{A}$ we can associate the set A of elements of S for which $\mathcal{A}$ is true while
- with any subset T of S we can associate the statement "$t \in T$."

Here are some simple, useful properties of Pr. You should be able to supply the proofs by writing all probabilities as sums of probabilities of elementary events and noticing which elementary events appear in which sums.

Theorem C.1 *Suppose $(S, \Pr)$ is a probability space and A, $A_1, \ldots, A_k$ and B are subsets of S.*

(a) *If $A \subseteq B$, then $0 \le \Pr(A) \le \Pr(B) \le 1$.*

(b) $\Pr(S \setminus A) + \Pr(A) = 1$. *One also writes $S - A$, A^c and A' for $S \setminus A$.*

(c) $\Pr(A \cup B) \;=\; \Pr(A) + \Pr(B) - \Pr(A \cap B)$.

(d) $\Pr(A_1 \cup \cdots \cup A_k) \;\le\; \Pr(A_1) + \cdots + \Pr(A_k)$.

We now define a "random variable." It is neither a variable nor random, rather, it is a function:

Definition C.2 Random variable *Given a probability space $(S, \Pr)$, a* **random variable** *on the space is a function $X : S \to \mathbb{R}$.*

People often use capital letters near the end of the alphabet to denote random variables. Why the name "random variable" for a function? The terminology arose historically before probability theory was put on a mathematical foundation. For example, suppose you toss a coin 10 times and let X be the number of heads. The value of X varies with the results of your tosses and it is random because your tosses are random. In probability theory terms, if the coin tosses are fair, we can define a probability space and a random variable as follows.

- S is the set of all 2^{10} possible 10-long head-tail sequences,
- $\Pr(s) = 1/|S| = 2^{-10} = 1/1024$,
- $X(s)$ equals the number of heads in the 10-long sequence s.

The probability that you get exactly four heads can be written as $\Pr(X = 4)$, which equals $\binom{10}{4}2^{-10}$ since there are $\binom{10}{4}$ 10-long head-tail sequences that contain exactly four heads.

Definition C.3 Independence *Let $(S, \Pr)$ be a probability space and let $\mathcal{X}$ be a set of random variables on $(S, \Pr)$. We say that the random variables in $\mathcal{X}$ are* **mutually independent** *if, for every subset $\{X_1, \ldots, X_k\}$ of $\mathcal{X}$ and all real numbers $x_1, \ldots, x_k$, we have*

$$\Pr(X_1 = x_1 \text{ and } \cdots \text{ and } X_k = x_k) \;=\; \Pr(X_1 = x_1) \cdots \Pr(X_k = x_k),$$

where the probabilities on the right are multiplied. We often abbreviate "mutual independence" to "independence."

Intuitively, the concept of independence means that knowing the values of some of the random variables gives no information about the values of others. For example, consider tossing a fair coin randomly ten times to produce a 10-long sequence of heads and tails. Define

$$X_i \;=\; \begin{cases} 1, & \text{if toss } i \text{ is heads,} \\ 0, & \text{if toss } i \text{ is tails.} \end{cases} \tag{C.1}$$

Then the set $\{X_1, \ldots, X_{10}\}$ of random variables are mutually independent.

We now look at "product spaces." They arise in natural ways and lead naturally to independence.

Definition C.4 Product space *Let $(S_1, \Pr_1), \ldots, (S_n, \Pr_n)$ be probability spaces. The product space*

$$(S, \Pr) \;=\; (S_1, \Pr_1) \times \cdots \times (S_n, Pr_n) \tag{C.2}$$

is defined by $S = S_1 \times \cdots \times S_n$, a Cartesian product, and

$$\Pr((a_1, \ldots, a_n)) \;=\; \Pr_1(a_1) \cdots \Pr_n(a_n) \quad \textit{for all} \quad (a_1, \ldots, a_n) \in S.$$

We may write $\Pr(a_1, \ldots, a_n)$ instead of $\Pr((a_1, \ldots, a_n))$.

As an example, consider tossing a fair coin randomly ten times. The probability space for a single toss is the set $\{H, T\}$, indicating heads and tails, with the uniform distribution. The product of ten copies of this space is the probability space for ten random tosses of a fair coin.

Theorem C.2 Independence in product spaces *Suppose a product space $(S, \Pr)$ is given by (C.2). Let $I_1, \ldots, I_m$ be pairwise disjoint subsets of $\underline{n}$; that is, $I_i \cap I_j = \emptyset$ whenever $i \neq j$. Suppose X_k is a random variable whose value on $(a_1, \ldots, a_n)$ depends only on the values of those a_i for which $i \in I_k$. Then $X_1, \ldots, X_m$ are mutually independent.*

We omit the proof.

Continuing with our coin toss example, Let $I_i = \{i\}$ and define X_i by (C.1). By the theorem, the X_i are mutually independent.

C.2 Expectation and Variance

Definition C.5 Expectation *Let $(S, \Pr)$ be a probability space The* **expectation of a random variable** *X is*

$$\mathbf{E}(X) \;=\; \sum_{s \in S} X(s) \Pr(s).$$

The expectation is also called the **mean**.

Let $R \subset \mathbb{R}$ be the set of values taken on by X. By collecting terms in the sum over S according to the value of $X(s)$ we can rewrite the definition as

$$\mathbf{E}(X) \;=\; \sum_{r \in R} r \Pr(X = r).$$

In this form, it should be clear the expected value of a random variable can be thought of as its average value.

Definition C.6 Variance *Let $(S, \Pr)$ be a probability space The* **variance of a random variable** *X is*

$$\mathbf{var}(X) \;=\; \sum_{s \in S} (X(s) - \mathbf{E}(X))^2 \Pr(s) \;=\; \sum_{r \in R} (r - \mathbf{E}(X))^2 \Pr(X = r),$$

where $R \subset \mathbb{R}$ is the set of values taken on by X.

The variance of a random variable measures how much it tends to deviate from its expected value. One might think that

$$\sum_{r \in R} |r - \mathbf{E}(X)| \Pr(X = r)$$

would be a better measure; however, there are computational and theoretical reasons for preferring the variance.

We often have a random variable X that takes on only the values 0 and 1. Let $p = \Pr(X = 1)$. You should be able to prove that $\mathbf{E}(X) = p$ and $\mathbf{var}(X) = p(1 - p)$.

The following theorem can be proved be algebraic manipulations of the definitions of mean and variance. Since this is an appendix, we omit the proof.

Theorem C.3 Properties of Mean and Variance *Let $X_1, \ldots, X_k$ and Y be random variables on a probability space $(S, \Pr)$.*

(a) *For real numbers a and b, $\mathbf{E}(aY + b) = a\,\mathbf{E}(Y) + b$ and $\mathbf{var}(aY + b) = a^2\,\mathbf{var}(Y)$.*

(b) $\mathbf{var}(Y) \;=\; \mathbf{E}\big((Y - \mathbf{E}(Y))^2\big) \;=\; \mathbf{E}(Y^2) - \big(\mathbf{E}(Y)\big)^2$.

(c) $\mathbf{E}(X_1 + \cdots + X_k) \;=\; \mathbf{E}(X_1) + \cdots + \mathbf{E}(X_k)$.

(d) *If the X_i are independent, then $\mathbf{var}(X_1 + \cdots + X_k) = \mathbf{var}(X_1) + \cdots + \mathbf{var}(X_k)$.*

Chebyshev's Theorem tells us that it is unlikely that the value of a random variable will be far from its mean, where the "unit of distance" is the square root of the variance. (The square root of $\mathbf{var}(X)$ is also called the *standard deviation* of X and is written σ_X.)

Theorem C.4 Chebyshev's inequality *If X is a random variable on a probability space and $t \geq 1$, then*

$$\Pr\Big(|X - \mathbf{E}(X)| > t\sqrt{\mathbf{var}(X)}\Big) \;<\; \frac{1}{t^2}. \tag{C.3}$$

For example, if a fair coin is tossed n times and X is the number of heads, then one can show that

$$\mathbf{E}(X) = n/2 \quad \text{and} \quad \mathbf{var}(X) = n/4. \tag{C.4}$$

Chebyshev's inequality tells us that X is not likely to be many multiples of $\sqrt{n}$ from $n/2$. Specifically, it says that

$$\Pr\Big(|X - n/2| > (t/2)\sqrt{n}\Big) \;<\; \frac{1}{t^2}.$$

Let's use Theorem C.3 to prove (C.4). Let Y_k be a random variable which is 1 if toss k lands heads and is 0 if it lands tails. By the definition of mean,

$$\mathbf{E}(Y_k) \;=\; 0\Pr(Y_k = 0) + 1\Pr(Y_k = 1) \;=\; 0 + 1/2 \;=\; 1/2$$

and, by the Theorem C.3(b) and the observation that $Y_k^2 = Y_k$,

$$\mathbf{var}(Y_k) \;=\; \mathbf{E}(Y_k^2) - \mathbf{E}(Y_k)^2 \;=\; \mathbf{E}(Y_k) - \mathbf{E}(Y_k)^2 \;=\; 1/2 - (1/2)^2 \;=\; 1/4.$$

Notice that $X = Y_1 + Y_2 + \cdots + Y_n$ and the Y_k are independent since the coin is tossed randomly. By Theorem C.3(c) and (d), we have (C.4).

APPENDIX D

Partial Fractions

We will discuss those aspects of partial fractions that are most relevant in enumeration. Although not necessary for our purposes, the theoretical background of the subject consists of two easily discussed parts, so we include it. The rest of this appendix is devoted to computational aspects of partial fractions.

Theory

The following result has many applications, one of which is to the theory of partial fractions.

Theorem D.1 Fundamental Theorem of Algebra *If $p(x)$ is a polynomial of degree n whose coefficients are complex numbers, then $p(x)$ can be written as a product of linear factors, each of which has coefficients which are complex numbers.*

We will not prove this.

In calculus classes, one usually uses a corollary of this theorem: If the coefficients of $p(x)$ are real numbers, then it is possible to factor $p(x)$ into a product of linear and quadratic factors, each of which has real coefficients. We will not use the corollary because it is usually more useful in combinatorics to write $p(x)$ as a product of linear factors.

By the Fundamental Theorem of Algebra, every polynomial $p(x)$ can be factored in the form

$$p(x) \;=\; Cx^n(1-\alpha_1 x)^{n_1}(1-\alpha_2 x)^{n_2}\cdots(1-\alpha_k x)^{n_k}, \tag{D.1}$$

where the α_i's are distinct nonzero complex numbers. Although this can always be done, it is, in general, *very difficult* to do. In (D.1), the α_i's are the reciprocals of the nonzero roots of $p(x)=0$ and n_i is the "multiplicity" of the root $1/\alpha_i$.

Suppose that $p(x)=p_1(x)p_2(x)\cdots p_k(x)$ and $q(x)$ are polynomials such that

- the degree of $q(x)$ is less than the degree of $p(x)$;
- none of the $p_i(x)$ is a constant;
- no pair $p_i(x)$ and $p_j(x)$ have a common root.

The Chinese Remainder Theorem for polynomials asserts that there exist unique polynomials $q_1(x),\ldots,q_k(x)$ (depending on $q(x)$ and the $p_i(x)$) such that

- the degree of $q_i(x)$ is less than the degree of $p_i(x)$ for all i;
- if the coefficients of $q(x)$ and the $p_i(x)$ are rational, then so are the coefficients of the $q_i(x)$;
- $\dfrac{q(x)}{p(x)} = \dfrac{q_1(x)}{p_1(x)} + \dfrac{q_2(x)}{p_2(x)} + \cdots + \dfrac{q_k(x)}{p_k(x)}$.

This is called a *partial fraction expansion* of $q(x)/p(x)$. For combinatorial applications, we take the $p_i(x)$'s to be of the form $(1-\alpha_i x)^{n_i}$.

Suppose we have been able to factor $p(x)$ as shown in (D.1). In our applications, we normally have $n = 0$, so we will assume this is the case. We can also easily remove the factor of C by dividing $q(x)$ by C. Let $p_i(x) = (1-\alpha_i x)^{n_i}$. With some work, we can obtain a partial fraction expansion for $q(x)/p(x)$. Finally, with a bit more work, we can rewrite $q_i(x)/(1-\alpha_i x)^{n_i}$ as

$$\frac{\beta_{i,1}}{1-\alpha_i x} + \frac{\beta_{i,2}}{(1-\alpha_i x)^2} + \cdots + \frac{\beta_{i,n_i}}{(1-\alpha_i x)^{n_i}},$$

where the $\beta_{i,j}$'s are complex numbers. (We will not prove this.)

Authors of calculus texts prefer a different partial fraction expansion for $q(x)/p(x)$. In the first place, as we already noted, they avoid complex numbers. This can be done by appropriately combining factors in (D.1). In the second place, they usually prefer that the highest degree term of each factor have coefficient 1, unlike combinatorialists who prefer that the constant term be 1.

Computations

The computational aspect of partial fractions has two parts. The first is the factoring of a polynomial $p(x)$ and the second is obtaining a partial fraction expansion.

In general, factoring is difficult. The polynomials we deal with can be factored by using the factoring methods of basic algebra, including the formula for the roots of a quadratic equation. The latter is used as follows:

$$Ax^2 + Bx + C \;=\; A(x-r_1)(x-r_2) \quad \text{where} \quad r_1, r_2 \;=\; \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}.$$

We will not review basic algebra methods. There is one unusual aspect to the sort of factoring we want to do in connection with generating functions. We want to factor $p(x)$ so that it is a product of a constant and factors of the form $1 - cx$. This can be done by factoring the polynomial $p(1/y)y^n$, where n is the degree of $p(x)$. The examples should make this clearer.

Example D.1 A factorization Factor the polynomial $p(x) = 1 - x - 4x^3$. Following the suggestion, we look at

$$r(y) \;=\; p(1/y)y^3 = y^3 - y^2 - 4.$$

Since $r(2) = 0$, $y - 2$ must be a factor of $r(y)$. Dividing it out we obtain $y^2 + y + 2$. By the quadratic formula, the zeroes of $y^2 + y + 2$ are $(-1 \pm \sqrt{-7}\,)/2$. Thus

$$r(y) \;=\; \Big(y-2\Big)\Big(y - \frac{-1+\sqrt{-7}}{2}\Big)\Big(y - \frac{-1-\sqrt{-7}}{2}\Big).$$

Since $p(x) = x^3 r(1/x)$, we finally have

$$p(x) \;=\; \Big(1-2x\Big)\Big(1 - \frac{-1+\sqrt{-7}}{2}x\Big)\Big(1 - \frac{-1-\sqrt{-7}}{2}x\Big). \quad \square$$

Example D.2 Partial fractions for a general quadratic We want to expand $q(x)/p(x)$ in partial fractions when $p(x)$ is a quadratic with distinct roots and $q(x)$ is of degree one or less. We will assume that $p(x)$ has been factored:

$$p(x) \;=\; (1-ax)(1-bx).$$

Since $p(x)$ has distinct roots, $a \neq b$.

Let us expand $1/p(x)$. We can write

$$\frac{1}{(1-ax)(1-bx)} \;=\; \frac{u}{1-ax} + \frac{v}{1-bx}, \tag{D.2}$$

where u and v are numbers that we must find. (If a and b are real, then u and v will be, too; however, if a or b is complex, then u and v could also be complex.) There are various ways we could find u and v. We'll show you two methods.

The straightforward method is to clear (D.2) of fractions and then equate powers of x. Here's what happens: Since

$$1 \;=\; u(1-bx) + v(1-ax) = (u+v) - (bu+av)x,$$

we have

$$1 \;=\; u+v \qquad \text{and} \qquad 0 \;=\; -(bu+av).$$

The solution to these equations is $u \;=\; a/(a-b)$ and $v \;=\; b/(b-a)$.

Another method for solving (D.2) is to multiply it by $1-ax$ and then choose x so that $1-ax=0$; i.e., $x = 1/a$. After the first step we have

$$\frac{1}{1-bx} \;=\; u + \frac{v(1-ax)}{1-bx}. \tag{D.3}$$

When we set $1-ax=0$, the last term in (D.3) disappears—that's why we chose that value for x. Substituting in (D.3), we get

$$u \;=\; \frac{1}{1-b/a} = \frac{a}{a-b}.$$

By the symmetry of (D.2), v is obtained simply by interchanging a and b.

We have shown by two methods that

$$\frac{1}{(1-ax)(1-bx)} \;=\; \frac{\frac{a}{a-b}}{1-ax} + \frac{\frac{b}{b-a}}{1-bx}. \tag{D.4}$$

By either of these methods, one can show that

$$\frac{x}{(1-ax)(1-bx)} \;=\; \frac{\frac{1}{a-b}}{1-ax} + \frac{\frac{1}{b-a}}{1-bx}. \tag{D.5}$$

We leave this as an exercise. You can save yourself quite a bit of work in partial fraction calculations if you use (D.4) and (D.5). □

Example D.3 A specific quadratic

Let's expand

$$\frac{1+3x}{(1+2x)(1-3x)}$$

by partial fractions. Using (D.4) and (D.5) with $a=-2$ and $b=3$, we easily obtain

$$\begin{aligned}\frac{1+3x}{(1+2x)(1-3x)} &= \frac{1}{(1+2x)(1-3x)}+\frac{3x}{(1+2x)(1-3x)}\\ &= \frac{2/5}{1+2x}+\frac{3/5}{1-3x}+\frac{-3/5}{1+2x}+\frac{3/5}{1-3x}\\ &= \frac{-1/5}{1+2x}+\frac{6/5}{1-3x}.\end{aligned} \tag{D.6}$$

To see how much effort has been saved, you are encouraged to derive this result without using (D.4) and (D.5). □

Example D.4 A factored cubic

Let's expand

$$\frac{1+3x}{(1-x)(1+2x)(1-3x)}$$

by partial fractions. We begin by factoring out $1-x$ and using (D.6). Next we use some algebra and then apply (D.4) twice. Here it is.

$$\begin{aligned}\frac{1+3x}{(1-x)(1+2x)(1-3x)} &= \frac{1}{1-x}\left(\frac{-1/5}{1+2x}+\frac{6/5}{1-3x}\right)\\ &= \frac{-1/5}{(1-x)(1+2x)}+\frac{6/5}{(1-x)(1-3x)}\\ &= \frac{(-1/5)(1/3)}{1-x}+\frac{(-1/5)(2/3)}{1+2x}+\frac{(6/5)(-1/2)}{1-x}+\frac{(6/5)(3/2)}{1-3x}\\ &= \frac{-2/3}{1-x}+\frac{-2/15}{1+2x}+\frac{9/5}{1-3x}.\end{aligned}$$

Notice how we were able to deal with the cubic denominator by iterating the method for dealing with a quadratic denominator. This will work in any situation as long as the denominator has no repeated factors. □

Example D.5 A squared quadratic

Let's expand

$$\frac{1+3x}{(1+2x)^2(1-3x)^2}. \tag{D.7}$$

Before tackling this problem, let's do the simpler case where the numerator is one:

$$\frac{1}{(1+2x)^2(1-3x)^2} = \left(\frac{1}{(1+2x)(1-3x)}\right)^2. \tag{D.8}$$

Using (D.4), this becomes

$$\left(\frac{2/5}{1+2x}+\frac{3/5}{1-3x}\right)^2,$$

which can be expanded to

$$\frac{4/25}{(1+2x)^2}+\frac{9/25}{(1-3x)^2}+\frac{12/25}{(1+2x)(1-3x)}.$$

The first two terms are already in standard partial fraction form and you should have no trouble expanding the last.

How does this help us with (D.7) since we still have a $3x$ left in the numerator? We can cheat a little bit and allow ourselves to write the expansion of (D.7) as simply $1+3x$ times the expansion of (D.8). This does not cause any problems in the applications of partial fractions that we are interested in other than slightly more complicated answers. □

Example D.6 Another problem How can we expand $1/(1-x)^2(1-2x)$ by partial fractions? Do any of our earlier tricks work? Yes, we simply write

$$\frac{1}{(1-x)^2(1-2x)} \;=\; \frac{1}{1-x}\left(\frac{1}{(1-x)(1-2x)}\right)$$

and continue as in Example D.4. □

As you should have seen by now, a bit of cleverness with partial fractions can save quite a bit of work. Another trick worth remembering is to use letters in place of complicated numbers. We conclude with an example which illustrates another trick.

***Example D.7 More tricks** Expand

$$\frac{x+6x^2}{1-x-4x^3} \tag{D.9}$$

in partial fractions.

We'll write the denominator as

$$(1-2x)(1+x+2x^2) \;=\; (1-2x)(1-cx)(1-dx)$$

where

$$c \;=\; \frac{-1+\sqrt{-7}}{2} \qquad \text{and} \qquad d \;=\; \frac{-1-\sqrt{-7}}{2},$$

a factorization found in Example D.1.

There are various ways we can attack this problem. Let's think about this.

- An obvious approach is to use the method of Example D.4. The $6x^2$ causes a bit of a problem in the numerator because the first step in expanding

  $$\frac{x+6x^2}{1+x+2x^2}$$

  is to divide so that the numerator is left as a lower degree than the denominator. If we take this approach, we should carry along c and d as much as possible rather than their rather messy values. Also, since they are zeros of y^2+y+2, we have the equations $c^2=-c-2$ and $d^2=-d-2$, which may help simplify some expressions. Also, $cd=2$ and $c+d=-1$. We leave this approach for you to carry out.
- We could use the previous idea after first removing the factor of $x+6x^2$ and then reintroducing it at the end. The justification for this is the last paragraph of Example D.5.
- Another approach is to write $x+6x^2 \;=\; x(1+6x)$. We can obtain partial fractions for $(1+6x)/(1+x+x^2)$ using (D.4) and (D.5). This result can be multiplied by $x/(1-2x)$ and expanded by (D.5).
- A different approach is to remove the $1-2x$ partial fraction term from (D.9), leaving a quadratic denominator. The resulting problem can then be done by our trusty formulas (D.4) and (D.5). Let's do this. We have

  $$\frac{x+6x^2}{(1-2x)(1+x+2x^2)} \;=\; \frac{u}{1-2x}+\frac{v}{1-cx}+\frac{w}{1-dx}. \tag{D.10}$$

Applying the trick of multiplying by $1-2x$ and setting $1-2x=0$, we have

$$u = \left.\frac{x+6x^2}{1+x+2x^2}\right|_{x=1/2} = 1.$$

Subtracting $u/(1-2x)$ from both sides of (D.10), we obtain

$$\begin{aligned}\frac{v}{1-cx}+\frac{w}{1-dx} &= \frac{x+6x^2}{(1-2x)(1+x+2x^2)}-\frac{1}{1-2x}\\ &= \frac{-1+4x^2}{(1-2x)(1+x+2x^2)}\\ &= \frac{-1-2x}{1+x+2x^2}.\end{aligned}$$

The last of these can now be expanded by (D.4) and (D.5). The cancellation of a factor of $1-2x$ from the numerator and denominator was not luck. It had to happen if our algebra was correct because we were removing the $1-2x$ partial fraction term. □

Since computing partial fractions can involve a lot of algebra, it is useful to have an algebra package do the computations. If that's not feasible, it's a good idea to check your calculations. This can be done in various ways

- Use a graphing calculator to plot the partial fraction expansion and the original fraction and see if they agree.
- Reverse the procedure: combine the partial fractions into one fraction and see if the result equals the original fraction.
- Compute several values to see if the value of the partial fraction expansion agrees with the value of the original fraction. For $p(x)/q(x)$, it suffices to compute $\max(\deg(q(x)),\ \deg(p(x))+1)$ values.

If you wish to practice working problems involving partial fractions, look in the partial fractions section of any complete calculus text.

Section 1.1

1.1.1. We can form n digit numbers by choosing the leftmost digit AND choosing the next digit AND $\cdots$ AND choosing the rightmost digit. The first choice can be made in 9 ways since a leading zero is not allowed. The remaining $n-1$ choices can each be made in 10 ways. By the Rule of Product we have $9 \times 10^{n-1}$.

To count numbers with at most n digits, we could sum up $9 \times 10^{k-1}$ for $1 \leq k \leq n$. The sum can be evaluated since it is a geometric series. This does not include the number 0. Whether we add 1 to include it depends on our interpretation of the problem's requirement that there be no leading zeroes. There is an easier way. We can pad out a number with less than n digits by adding leading zeroes. The original number can be recovered from any such n digit number by stripping off the leading zeroes. Thus we see by the Rule of Product that there are 10^n numbers with at most n digits. If we wish to rule out 0 (which pads out to a string of n zeroes), we must subtract 1.

1.1.3. List the elements of the set in any order: $a_1, a_2, \ldots, a_{|S|}$. We can construct a subset by

including a_1 or not AND
including a_2 or not AND
. . .
including $a_{|S|}$ or not.

Since there are 2 choices in each case, the Rule of Product gives $2 \times 2 \times \cdots \times 2 = 2^{|S|}$.

1.1.5. The answers are SISITS and SISLAL. We'll come back to this type of problem when we study decision trees.

Section 1.2

1.2.1. If we want all assignments of birthdays to people, then repeats are allowed in the list mentioned in the hint. This gives 365^{30}. If we want all birthdays distinct, no repeats are allowed in the list. This gives $365 \times 364 \times \cdots \times (365-29)$. The ratio is 0.29. How can this be computed? There are a lot of possibilities. Here are some.

- Use a symbolic math package.
- Write a computer program.
- Use a calculator. Overflow may be a problem, so you might write the ratio as $(365/365) \times (364/365) \times \cdots \times (336/365)$.
- Use (1.2). You are asked to do this in the next problem. Unfortunately, there is no guarantee how large the error will be.
- Use Stirling's formula after writing the numerator as 365!/335!. Since Stirling's formula has an error guarantee, we know we are close enough. Computing the values directly from Stirling's formula may cause overflow. This can be avoided in various ways. One is to rearrange the various factors by using some algebra:

$$\frac{\sqrt{2\pi 365}(365/e)^{365}}{\sqrt{2\pi 335}(335/e)^{335}(365)^{30}} = \sqrt{365/335}\,(365/335)^{335}/e^{30}.$$

Another way is to compute the logarithm of Stirling's formula and use that to estimate the logarithm of the answer.

1.2.3. Each of the 7 letters ABMNRST appears once and each of the letters CIO appears twice. Thus we must form an ordered list from the 10 distinct letters. The solutions are

$$\begin{aligned} k=2: \quad & 10 \times 9 = 90 \\ k=3: \quad & 10 \times 9 \times 8 = 720 \\ k=4: \quad & 10 \times 9 \times 8 \times 7 = 5040 \end{aligned}$$

1.2.5 (a) Since there are 5 distinct letters, the answer is $5 \times 4 \times 3 = 60$.

(b) Since there are 5 distinct letters, the answer is $5^3 = 125$.

(c) Either the letters are distinct OR one letter appears twice OR one letter appears three times. We have seen that the first can be done in 60 ways. To do the second, choose one of L and T to repeat, choose one of the remaining 4 different letters and choose where that letter is to go, giving $2 \times 4 \times 3 = 24$. To do the third, use T. Thus, the answer is $60 + 24 + 1 = 85$.

1.2.7 (a) push, push, pop, pop, push, push, pop, push, pop, pop. Remembering to start with something, say a on the stack: $(a(bc))((de)f)$.

(b) This is almost the same as (a). The sequence is 112211212122 and the last "pop" in (a) is replaced by "push, pop, pop."

(c) $a((b((cd)e))(fg))$; push, push, push, pop, push, pop, pop, push, push, pop, pop, pop; 111010011000.

1.2.9. Stripping off the initial R and terminal F, we are left with a list of at most 4 letters, at least one of which is an L. There is just 1 such list of length 1. There are $3^2 - 2^2 = 5$ lists of length 2, namely all those made from E, I and L minus those made from just E and I. Similarly, there are $3^3 - 2^3 = 19$ of length 3 and $3^4 - 2^4 = 65$. This gives us a total of 90.

The letters used are E, F, I, L and R in alphabetical order. To get the word before RELIEF, note that we cannot change just the F and/or the E to produce an earlier word. Thus we must change the I to get the preceding word. The first candidate in alphabetical order is F, giving us RELF. Working backwards in this manner, we come to RELELF, RELEIF, RELEF and, finally, RELEEF.

1.2.11. There are $n!/(n-k)!$ lists of length k. The total number of lists (not counting the empty list) is

$$\frac{n!}{(n-1)!} + \frac{n!}{(n-2)!} + \cdots + \frac{n!}{1!} + \frac{n!}{0!} \;=\; n!\left(\frac{1}{0!} + \frac{1}{1!} + \cdots + \frac{1}{(n-1)!}\right) \;=\; n!\sum_{i=0}^{n-1}\frac{1^i}{i!}.$$

Since $e = e^1 = \sum_{i=0}^{\infty} 1^i/i!$, it follows that the above sum is close to e.

1.2.13. We can only do parts (a) and (d) at present.

(a) A person can run for one of k offices or for nothing, giving $k+1$ choices per person. By the Rule of Product we get $(k+1)^p$.

(d) We can treat each office separately. There are $2^p - 1$ possible slates for an office: any subset of the set of candidates except the empty one. By the Rule of Product we have $(2^p - 1)^k$.

Section 1.3

1.3.1. After recognizing that $k = n\lambda$ and $n - k = n(1 - \lambda)$, it's simply a matter of algebra.

1.3.3. Choose values for pairs AND choose suits for the lowest value pair AND choose suits for the middle value pair AND choose suits for the highest value pair. This gives $\binom{13}{3}\binom{4}{2}^3 = 61,776$.

1.3.5. Choose the lowest value in the straight (A to 10) AND choose a suit for each of the 5 values in the straight. This gives $10 \times 4^5 = 10240$.

Although the previous answer is acceptable, a poker player may object since a "straight flush" is better than a straight—and we included straight flushes in our count. Since a straight flush is a straight all in the same suit, we only have 4 choices of suits for the cards instead of 4^5. Thus, there are $10 \times 4 = 40$ straight flushes. Hence, the number of straights which are not straight flushes is $10240 - 40 = 10200$.

1.3.7. This is like Exercise 1.2.3, but we'll do it a bit differently. Note that EXERCISES contains 3 E's, 2 S's and 1 each of C, I, R and X. By the end of Example 1.18, we can use (1.4) with $N = 9$, $m_1 = 3$, $m_2 = 2$ and $m_3 = m_4 = m_5 = m_6 = 1$. This gives $9!/3!\,2! = 30240$.

It can also be done without the use of a multinomial coefficient as follows. Choose 3 of the 9 possible positions to use for the three E's AND choose 2 of the 6 remaining positions to use for the two S's AND put a permutation of the remaining 4 letters in the remaining 4 places. This gives us $\binom{9}{3} \times \binom{6}{2} \times 4!$.

The number of eight letter arrangements is the same. To see this, consider a 9-list with the ninth position labeled "unused."

1.3.9. Think of the teams as labeled and suppose Teams 1 and 2 each contain 3 men. We can divide the men up in $\binom{11}{3,3,2,2,1}$ ways and the women in $\binom{11}{2,2,3,3,1}$ ways.

We must now count the number of ways to form the ordered situation from the unordered one. Be careful—it's not $4! \times 2$ as it was in the example! Thinking as in the early card example, we start out two types of teams, say M or F depending on which sex predominates in the team. We also have two types of referees. Thus we have two M teams, two F teams, and one each of an F referee and an M referee. We can order the two M teams (2 ways) and the two F teams (2 ways), so there are only 2×2 ways to order and so the answer is $\binom{11}{3,3,2,2,1}^2 \frac{1}{4}$.

1.3.11. The theorem is true when $k = 2$ by the binomial theorem with $x = y_1$ and $y = y_2$. Suppose that $k > 2$ and that the theorem is true for $k - 1$. Using the hint and the binomial theorem with $x = y_k$ and $y = y_1 + y_2 + \cdots + y_{k-1}$, we have that

$$(y_1 + y_2 + \cdots + y_k)^n = \sum_{j=0}^{n} \binom{n}{j} (y_1 + y_2 + \cdots + y_{k-1})^{n-j} y_k^j.$$

Thus the coefficient of $y_1^{m_1} \cdots y_k^{m_k}$ in this is $\binom{n}{m_k} = n!/(n - m_k)!m_k!$ times the coefficient of $y_1^{m_1} \cdots y_{k-1}^{m_{k-1}}$ in $(y_1 + y_2 + \cdots + y_{k-1})^{n-m_k}$. When $n - m_k = m_1 + m_2 + \cdots + m_{k-1}$ the coefficient is $(n - m_k)!/m_1!m_2! \cdots m_{k-1}!$ and otherwise it is zero by the induction assumption. Multiplying by $\binom{n}{m_k}$, we obtain the theorem for k.

Section 1.4

1.4.1. The rows are 1,7,21,35,35,7,1 and 1,8,28,56,70,56,28,8,1.

1.4.3. Let $L(n,k)$ be the number of ordered k-lists without repeats that can be made from an n-set S. Form such a list by choosing the first element AND then forming a $k-1$ long list using the remaining $n-1$ elements. This gives $L(n,k) = nL(n-1,k-1)$.

Single out one item $x \in S$. There are $L(n-1,k)$ lists not containing x. If x is in the list, it can be in any of k positions AND the rest of the list can be constructed in $L(n-1,k-1)$ ways. Thus

$$L(n,k) \;=\; L(n-1,k) + kL(n-1,k-1).$$

1.4.5. The only way to partition an n element set into n blocks is to put each element in a block by itself, so $S(n,n) = 1$. The only way to partition an n element set into one block is to put all the elements in the block, so $S(n,1) = 1$.

The only way to partition an n element set into $n-1$ blocks is to choose two elements to be in a block together an put the remaining $n-2$ elements in $n-2$ blocks by themselves. Thus it suffices to choose the 2 elements that appear in a block together and so $S(n,n-1) = \binom{n}{2}$.

The formula for $S(n,n-1)$ can also be proved using (1.9) and induction. The formula is correct for $n = 1$ since there is no way to partition a 1-set and have no blocks. Assume true for $n-1$. Use the recursion, the formula for $S(n-1,n-1)$ and the induction assumption for $S(n-1,n-2)$ to obtain

$$S(n,n-1) = S(n-1,n-2) + (n-1)S(n-1,n-1) = \binom{n-1}{2} + (n-1)1 = \binom{n}{2},$$

which completes the proof.

Now for $S(n,2)$. Note that $S(n,k)$ is the number of unordered lists of length k where the list entries are nonempty subsets of a given n-set and each element of the set appears in exactly one list entry. We will count ordered lists, which is $k!$ times the number of unordered ones. We choose a subset for the first block (first list entry) and use the remaining set elements for the second block. Since an n-set has 2^n, this would seem to give $2^n/2$; however, we must avoid empty blocks. In the ordered case, there are two ways this could happen since either the first or second list entry could be the empty set. Thus, we must have $2^n - 2$ instead of 2^n.

Here is another way to compute $S(n,2)$. Look at the block containing n. Once it is determined, the entire two block partition is determined. The block one of the 2^{n-1} subsets of $\underline{n-1}$ with n adjoined. Since something must be left to form the second block, the subset cannot be all of $\underline{n-1}$. Thus there are $2^{n-1}-1$ ways to form the block containing n.

The formula for $S(n,2)$ can also be proved by induction using the recursion for $S(n,k)$ and the fact that $S(n,1) = 1$, much as was done for $S(n,n-1)$.

1.4.7. There are $\binom{n}{k}$ ways to choose the subset AND k ways to choose an element in it to mark. This gives the left side of the recursion times k. On the other hand, there are n ways to choose an element to mark from $\{1,2,\ldots,n\}$ AND $\binom{n-1}{k-1}$ ways to choose the remaining elements of the k-element subset.

1.4.9 (b) Each office is associated with a nonempty subset of the people and each person must be in exactly one subset. This is a partition of the set of candidates with each block corresponding to an office. Thus we have an ordered partition of a n element set into k blocks. The answer is $k!S(n,k)$.

(c) This is like the previous part, except that some people may be missing. We use two methods. First, let i people run for no offices. The remaining $n-i$ can be partitioned in $S(n-i,k)$ ways and the blocks ordered in $k!$ ways. Thus we get $\sum_{i\geq 0}\binom{n}{i}k!S(n-i,k)$. For the second method,

either everyone runs for an office, giving $k!S(n,k)$ or some people do not run. In the latter case, we can think of a partition with $k+1$ labeled blocks where the labels are the k offices and "not running." This give $(k+1)!S(n,k+1)$. Thus we have $k!S(n,k)+(k+1)!S(n,k+1)$. The last formula is preferable since it is easier to calculate from tables of Stirling numbers.

(e) Let $T(p,k)$ be the number of solutions. Look at all the people running for the first $k-1$ offices. Let t be the number of these people. If $t < p$, then at least $p-t$ people must be running for the kth office since everyone must run for some office. In addition, any of these t people could run for the kth office. By the Rule of Product, the number of ways we can have this particular set of t people running for the first $k-1$ offices and some people running for the kth office is $T(t,k-1)2^t$. The set of t people can be chosen in $\binom{p}{t}$ ways. Finally, look at the case $t=p$. In this case everyone is running for one of the first $k-1$ offices. The only restriction we must impose is that a nonempty set of candidates must run for the kth office. Putting all this together, we obtain

$$T(p,k) \;=\; \sum_{t=1}^{p-1} \binom{p}{t} T(t,k-1)2^t + T(p,k-1)(2^p-1).$$

This recursion is valid for $p \geq 2$ and $k \geq 2$. The initial conditions are $T(p,1)=1$ for $p>0$ and $T(1,k)=1$ for $k>0$.

Notice that if "people" and "offices" are interchanged, the problem is not changed. Thus $T(p,k)=T(k,p)$ and a recursion could have been obtained by looking at offices that the first $p-1$ people run for. This would give us

$$T(p,k) \;=\; \sum_{t=1}^{k-1} \binom{k}{t} T(p-1,t)2^t + T(p-1,k)(2^k-1).$$

Section 1.5

1.5.1. For each element, there are $j+1$ choices for the number of repetitions, namely anything from 0 to j, inclusive. By the Rule of Product, we obtain $(j+1)^{|S|}$.

1.5.3. To form an unordered list of length k with repeats from $\{1,2,\ldots,n\}$, either form a list without n OR form a list with n. The first can be done in $M(n-1,k)$ ways. The second can be done by forming a $k-1$ element list AND then adjoining n to it. This can be done in $M(n,k-1)\times 1$ ways. Initial conditions: $M(n,0)=1$ for $n \geq 0$ and $M(0,k)=0$ for $k>0$.

1.5.5. Interpret the points between the ith and the $(i+1)$st vertical bars as the balls in box i. Since there are $n+1$ bars, there are n boxes. Since there are $(n+k-1)-(n-1)=k$ points, there are k balls.

1.5.7. This exercise and the previous one are simply two different ways of looking at the same thing since an unordered list with repetitions allowed is the same as a multiset. The nth item must appear zero, one OR two times. The remaining $n-1$ items must be used to form a list of length k, $k-1$ or $k-2$ respectively. This gives the three terms on the left. We generalize to the case where each item is used at most j times: $T(n,k) \;=\; \sum_{i=0}^{j} T(n-1,k-i)$.

1.5.9 (a) We give two solutions. Both use the idea of inserting a ball into a tube in an arbitrary position. To physically do this may require some manipulation of balls already in the tube.

1. Insert $b-1$ balls into the tubes AND then insert the bth ball. There are $i+1$ possible places to insert this ball in a tube containing i balls. Summing this over all t tubes gives us $(b-1)+t$ possible places to insert the bth ball. We have proved that

$$f(b,t) \;=\; f(b-1,t)(b+t-1).$$

   Since $f(1,t)=t$, we can establish the formula by induction.

2. Alternatively, we can insert the first ball AND insert the remaining $b-1$ balls. The first ball has the effect of dividing the tube in which it is placed into two tubes: the part above it and the part below. Thus

$$f(b,t) \;=\; tf(b-1,t+1),$$

   and we can again use induction.

(b) We give two solutions:

Construct a list of length $t+b-1$ containing each ball exactly once and containing $t-1$ copies of "between tubes." This can be done in $\binom{t+b-1}{t-1}b!$ ways—choose the "between tubes" and then permute the balls to place them in the remaining b positions in the list.

Alternatively, imagine an ordered $b+t-1$ long list. Choose $t-1$ positions to be divisions between tubes AND choose how to place the b balls in the remaining b positions. This gives $\binom{b+t-1}{t-1}\times b!$.

Section 2.2

2.2.3. The interchanges can be written as (1,3), (1,4) and (2,3). Thus the entire set gives $1\to 3\to 2$, $2\to 3$, $3\to 1\to 4$ and $4\to 1$. In cycle form this is (1,2,3,4). Thus five applications takes 1 to 2.

2.2.5 (a) This was done in Exercise 2.2.2, but we'll redo it. If $f(k)=k$, then the elements of $\underline{n}-\{k\}$ can be permuted in any fashion. This can be done in $(n-1)!$. Since there are $n!$ permutations, the probability that $f(k)=k$ is $(n-1)!/n! = 1/n$. Hence the probability that $f(k)\neq k$ is $1-1/n$.

(b) By the independence assumption, the probability that there are no fixed points is $(1-1/n)^n$. One of the standard results in calculus is that this approaches $1/e$ as $n\to\infty$. (You can prove it by writing $(1-1/n)^n = \exp(\ln(1-1/n)/(1/n))$, setting $1/n=x$ and using l'Hôpital's Rule.)

(c) Choose the k fixed points AND construct a derangement of the remaining $n-k$. This gives us $\binom{n}{k}D_{n-k}$. Now use $D_{n-k}\approx (n-k)!/e$.

2.2.7. For $1\le k\le n-1$, $\mathbf{E}(|a_k-a_{k+1}|)=\mathbf{E}(|i-j|)$, where the latter expectation is taken over all $i\neq j$ in $\underline{n}$. Thus the answer is $(n-1)$ times the average of the $n(n-1)$ values of $|i-j|$ and so

$$\begin{aligned}
\text{answer} \;&=\; \frac{n-1}{n(n-1)}\sum_{i\neq j}|j-i| \;=\; \frac{n-1}{n(n-1)}\sum_{i,j}|j-i| \;=\; \frac{2}{n}\sum_{1\le i\le j\le n}(j-i), \qquad \text{proving (a)}\\
&=\; \frac{2}{n}\sum_{j=1}^{n}\sum_{i=1}^{j}(j-i) \;=\; \frac{2}{n}\sum_{j=1}^{n}\left(j^2-\frac{j(j+1)}{2}\right) \;=\; \frac{1}{n}\sum_{j=1}^{n}(j^2-j)\\
&=\; \frac{1}{n}\left(\frac{n(n+1)(2n+1)}{6}-\frac{n(n+1)}{2}\right) \;=\; \frac{n^2-1}{2}.
\end{aligned}$$

Section 2.3

2.3.3. We can form the permutations of the desired type by first constructing a partition of $\underline{n}$ counted by $B(n, \vec{b})$ AND then forming a cycle from each block of the partition. The argument used in Exercise 2.2.2 proves that there are $(k-1)!$ cycles of length k that can be made from a k-set.

2.3.5 (a) In the order given, they are 2, 1, 3 and 4

(b) If f is associated with a B partition of $\underline{n}$, then B is the coimage of f and so f determines B.

(c) See (b).

(d) The first is not since $f(1) = 2 \neq 1$.
The second is: just check the conditions.
The third is not since $f(4) - 1 = 2 > \max(f(1), f(2), f(3)) = 1$.
The fourth is: just check the conditions.

(e) In a way, this is obvious, but it is tedious to write out a proof. By definition $f(1) = 1$. Choose $k > 1$ such that $f(x) = k$ for some x. Let y be the least element of $\underline{n}$ for which $f(y) = k$. By the way f is constructed, y is not in the same block with any $t < y$. Thus y is the smallest element in its block and so $f(y)$ will be the smallest number exceeding all the values that have been assigned for $f(t)$ with $t < y$. Thus the maximum of $f(t)$ over $t < y$ is $k - 1$ and so f is a restricted growth function.

(f) The functions are given in one-line form and the partition below them

1 1 1 1	1 1 1 2	1 1 2 1	1 1 2 2	1 1 2 3
$\{1,2,3,4\}$	$\{1,2,3\}\,\{5\}$	$\{1,2,4\}\,\{3\}$	$\{1,2\}\,\{3,4\}$	$\{1,2\}\,\{3\}\,\{4\}$
1 2 1 1	1 2 1 2	1 2 1 3	1 2 2 1	1 2 2 2
$\{1,3,4\}\,\{2\}$	$\{1,3\}\,\{2,4\}$	$\{1,3\}\,\{2\}\,\{4\}$	$\{1,4\}\,\{2,3\}$	$\{1\}\,\{2,3,4\}$
1 2 2 3	1 2 3 1	1 2 3 2	1 2 3 3	1 2 3 4
$\{1\}\,\{2,3\}\,\{4\}$	$\{1,4\}\,\{2\}\,\{3\}$	$\{1\}\,\{2,4\}\,\{3\}$	$\{1\}\,\{2\}\,\{3,4\}$	$\{1\}\,\{2\}\,\{3\}\,\{4\}$

2.3.7. The coimage is a partition of A into at most $|B|$ blocks, so our bound is $1 + (|A| - 1)/|B|$.

2.3.9. If $s < t$ and $f(s) = f(t)$, that tells us that we cannot put a_s at the start of the longest decreasing subsequence starting with a_t to obtain a decreasing subsequence. (If we could, we'd have $f(s) \geq f(t) + 1$.) Thus, $a_s > a_t$. Hence the subsequence $a_i, a_j, \ldots$ constructed in the problem is increasing.

Now we're ready to start the proof. If there is a decreasing subsequence of length $n+1$ we are done. If there is no such subsequence, $f : \underline{\ell} \to \underline{n}$. By the generalized Pigeonhole Principle, there is sum k such that $f(t) = k$ for at least ℓ/n values of t. Thus it suffices to have $\ell/n > m$. In other words $\ell > mn$.

2.3.11. Let the elements be $s_1, \ldots, s_n$, let $t_0 = 0$ and let $t_i = s_1 + \ldots + s_i$ for $1 \leq i \leq n$. By the Pigeonhole Principle, two of the t's have the same remainder on division by n, say t_j and t_k with $j < k$. It follows that $t_k - t_j = s_{j+1} + \ldots + s_k$ is a multiple of n.

Section 2.4

2.4.1. $x(x+y) = xx + xy = x + xy = x$.

2.4.3. We state the laws and whether they are true or false. If false we give a counterexample.

(a) $x + (yz) = (x+y)(x+z)$ is true. (Proved in text.)

(b) $x(y \oplus z) = (xy) \oplus (xz)$ is true.

(c) $x + (y \oplus z) = (x+y) \oplus (x+z)$ is false with $x = y = z = 1$.

(d) $x \oplus (yz) = (x \oplus y)(x \oplus z)$ is false with $x = y = 1$, $z = 0$.

2.4.5. We use algebraic manipulation. Each step involves a simple formula, which we will not bother to mention. You could also write down the truth table, read off a disjunctive normal form and try to reduce the number of terms.

(a) $(x \oplus y)(x+y) = (xy' + x'y)(x+y) = xy' + x'yx + xy'y + x'y = xy' + x'y$. Note that this is $x \oplus y$.

(b) $(x+y) \oplus z = (x+y)z' + (x+y)'z = xz' + yz' + x'y'z$.

(c) $(x+y+z) \oplus z = (x+y+z)z' + (x+y+z)'z = xz' + yz' + x'y'z'z = xz' + yz'$.

(d) $(xy) \oplus z = xyz' + (xy)'z = xyz' + x'z + y'z$.

2.4.7. There are many possible answers. A complicated one comes directly from the truth table and contains 8 terms. The simplest form is $xw + yw + zw + xyz$. This can be obtained as follows. $(x+y+z)w$ will give the correct answer except when $x = y = z = 1$ and $w = 0$. Thus we could simply add the term $xyzw'$. By noting that it is okay to add xyz when $w = 1$, we obtain $(x+y+z)w + xyz$.

Section 3.1

3.1.1. From the figures in the text, we see that they are 123, 132 and 321.

3.1.3. We will not draw the tree. The root is 1, the vertices on the next level are 21 and 12 (left to right). On the next level, 321, 231, 213, 312, 132, and 123. Finally, the leaves are 4321, 3421, 3241, 3214, 4231, 2431, 2341, 2314, 4213, 2413, 2143, 2134, and so on.

(a) 7 and 16.

(b) 2,4,3,1 and 3,1,2,4.

3.1.5. We will not draw the tree. There are nine sequences: ABABAB, ABABBA, ABBABA, ABBABB, BABABA, BABABB, BABBAB, BBABAB and BBABBA.

3.1.7. We will not draw the tree.

(a) 5 and 18.

(b) 111 and 433.

(c) 4,4,4 has rank 19.

(e) The decision tree for the strictly decreasing functions is interspersed. To find it, discard the leftmost branch leading out of each vertex except the root and then discard those decisions that no longer lead to a leaf of the original tree.

3.1.9. We assume that you are looking at decision trees in the following discussion.

(a) The permutation of rank 0 is the leftmost one in the tree and so each element is inserted as far to the left as possible. Thus the answer is $n, (n-1), \ldots, 2, 1$.

The permutation of rank $n! - 1$ is the rightmost one in the tree and so each element is inserted as far to the right as possible. Thus the answer is $1, 2, 3, \ldots, n$.

We now look at $n!/2$. Note that the decision about where to insert 2 splits the tree into two equal pieces. We are interested in the leftmost leaf of the righthand piece. The righthand piece means we take the branch $1, 2$. To stay to the left after that, 3 through n are inserted in the leftmost position. Thus the permutation is $n, (n-1), \ldots, 4, 3, 1, 2$.

(b) The permutation of rank 0 is the leftmost one in the tree and so each element is inserted as far to the left as possible. It begins 2,1. Then 3 "bumps" 2 to the end: 3,1,2. Next 4 "bumps" 3 to the end: 4,1,2,3. In general, we have $n, 1, 2, 3, \ldots, (n-1)$.

The permutation of rank $n! - 1$ is the rightmost one in the tree and so each element is inserted as far to the right as possible. Thus the answer is $1, 2, 3, \ldots, n$.

We now look at $n!/2$. Note that the decision about where to insert 2 splits the tree into two equal pieces. We are interested in the leftmost leaf of the righthand piece. The righthand piece means we take the branch $1, 2$. To stay to the left after that, 3 through n are inserted in the leftmost position. This leads to "bumping" as it did for rank 0. Thus the permutation is $n, 2, 1, 3, 4, 5, \ldots, (n-1)$.

(c) You should be able to see that the permutation $(1, 2, 3, \ldots, n)$ has rank 0 in both cases and that the permutation $(n, \ldots, 3, 2, 1)$ has rank $n! - 1$ in both cases.

First suppose that $n = 2m$, an even number. It is easy to see how to split the tree in half based on the first decision as we did for insertion order: Choose $m + 1$ and then stay as left as possible. This means everything is in order except for $m + 1$. Thus the permutation is $m + 1$ followed by the elements of $\underline{n} - \{m + 1\}$ in ascending order.

Now suppose that $n = 2m - 1$. In this case, we must make the middle choice, m and split the remaining tree in half, going to the leftmost leaf of the right part. If you look at some trees, you should see that this leads to the permutation $m, m + 1$ followed by the elements of $\underline{n} - \{m, m + 1\}$ in ascending order.

3.1.11 (a) We'll make a decision based on whether or not the pair in the full house has the same face value as a pair in the second hand. If it does not, there are

$$\binom{11}{2}\binom{4}{2}^2(52 - 8 - 5) \;=\; 77{,}220$$

possible second hands. If it does, there are

$$11\binom{4}{2}(52 - 8 - 3) \;=\; 2{,}706$$

possible second hands. Adding these up and multiplying by the number of possible full houses (79,926) gives us about 3×10^8 hands.

(b) There are various ways to do this. The decision trees are all more complicated than in the previous part.

(c) The order in which things are done can be very important.

3.1.13. You can simply modify the decision tree in Figure 3.5 as follows: Decrease the "number of singles" values by 1 (since the desired word is one letter shorter). Throw away those that become negative; i.e., erase leaves C and H. Add a new path that has no triples, one pair and five singles.

Call the new leaf X. It is then necessary to recompute the numbers. Here are the results, which total to 113,540:

$$\begin{aligned}
X: &\quad \binom{2}{0}\binom{5}{1}\binom{5}{5}\binom{8}{2,\,1,\,1,\,1,\,1,\,1} = 12{,}600\\
A: &\quad \binom{2}{0}\binom{5}{2}\binom{4}{3}\binom{8}{2,\,2,\,1,\,1,\,1} = 50{,}400\\
B: &\quad \binom{2}{0}\binom{5}{3}\binom{3}{1}\binom{8}{2,\,2,\,2,\,1} = 18{,}900\\
D: &\quad \binom{2}{1}\binom{4}{0}\binom{5}{4}\binom{8}{3,\,1,\,1,\,1,\,1} = 8{,}400\\
E: &\quad \binom{2}{1}\binom{4}{1}\binom{4}{2}\binom{8}{3,\,2,\,1,\,1} = 20{,}160\\
F: &\quad \binom{2}{1}\binom{4}{2}\binom{3}{0}\binom{8}{3,\,2,\,2} = 2{,}520\\
G: &\quad \binom{2}{2}\binom{3}{0}\binom{4}{1}\binom{8}{3,\,3,\,1} = 560.
\end{aligned}$$

Section 3.2

3.2.1. We use the rank formula in the text and, for unranking, a greedy algorithm.

(a) $\binom{10}{3}+\binom{5}{2}+\binom{3}{1} = 133.$ $\quad \binom{8}{4}+\binom{5}{3}+\binom{2}{2}+\binom{0}{1} = 81.$

(b) We have $35 = \binom{7}{4}$ so the first answer is 8,3,2,1. The second answer is 12,9,6,5 because

$$\begin{aligned}
\tbinom{11}{4} \le 400 < \tbinom{12}{4} \qquad & 400 - \tbinom{11}{4} = 70\\
\tbinom{8}{3} \le 70 < \tbinom{9}{3} \qquad & 70 - \tbinom{8}{3} = 14\\
\tbinom{5}{2} \le 14 < \tbinom{6}{2} \qquad & 14 - \tbinom{5}{2} = 4\\
\tbinom{4}{1} \le 4 < \tbinom{5}{1}. &
\end{aligned}$$

(c) 9,6,4,2,1 and 9,7,2,1.

(d) 9,5,4,3,2 and 9,6,5,3.

3.2.3. One can compute the ranks by looking at the decision tree or by using the formula in Theorem 3.3. We choose the latter approach. In case (j), we have $f(i) = k + j - i$. (This is easily checked since this f clearly decreases by 1 as i increases by 1 and it gives $f(1) = k$, $k+1$ and $k+2$ for $j = 1$, 2 and 3, respectively.) By the theorem,

$$\text{RANK}(f) = \sum_{i=1}^{k} \binom{f(i)-1}{k+1-i} = \sum_{i=1}^{k} \binom{k+j-i-1}{k+1-i}$$

When $j = 1$, all the binomial coefficients are 0 and so the answer for the first function is 0.
When $j = 2$, all the binomial coefficients are 1 and so the answer for the second function is k.
When $j = 3$, we have

$$\text{RANK}(f) = \sum_{i=1}^{k} \binom{k+2-i}{k+1-i} = \sum_{i=1}^{k} (k+2-i) = (k+1)+(k)+(k-1)+\cdots+(2).$$

Since the sum of the first n positive integers is $\frac{n(n+1)}{2}$, the rank is $\frac{(k+1)(k+2)}{2} - 1 = \frac{k(k+3)}{2}$.

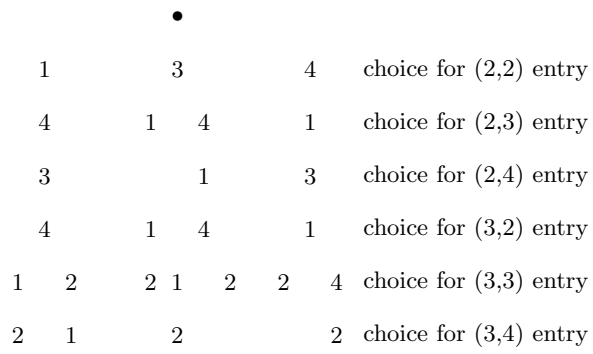

1 3 4 choice for (2,2) entry
4 1 4 1 choice for (2,3) entry
3 1 3 choice for (2,4) entry
4 1 4 1 choice for (3,2) entry
1 2 2 1 2 2 4 choice for (3,3) entry
2 1 2 2 choice for (3,4) entry

Figure S.3.1 The decision tree for 4×4 standard Latin Squares in Exercise 3.3.1.

3.2.5 (a) $D_1 \times (n-1)! + D_2 \times (n-2)! + \cdots + D_{n-1} \times 1! \;=\; \sum_{k=1}^{n-1} D_k(n-k)!$.

(b) Denote the permutation by f. Let $L = \underline{n}$. For $i = 1, 2, \ldots, n-1$ in order: let D_i is the number of elements in L which are less than $f(i)$ and replace L with $L - \{f(i)\}$.

(c) The decision sequences are 4,4,0,1,1 and 5,1,2,0,0 and so the ranks are 579 and 636.

(d) By a greedy algorithm we get the decision sequences 1,1,1,0,1 and 2,2,2,0,0. The permutations are 2,3,4,1,6,5 and 3,4,5,1,2,6.

3.2.9. $00000000000000000000 = 0^{20}$; $11000000000000000000 = 1^2 0^{18}$; 0100; 10101100.

Section 3.3

3.3.1. When building and $n \times n$ Latin Square, if the first $n-1$ rows have been filled in, then the last row is determined. Thus we'll omit it from the decision tree. The tree is shown in Figure S.3.1.

3.3.3. You should find 14 solutions.

Section 4.1

4.1.1. The Venn diagrams each consist of two intersecting circles.

(a) $V_2 \cap V_3$ contains words of the form $CVVC$. We are interested in $V_2 \cup V_3$, the union of the circles. Thus

$$\begin{aligned} |V_2 \cup V_3| &= |V_2| + |V_3| - |V_2 \cap V_3| \\ &= 21^2 \times 5 \times 26 + 21^2 \times 5 \times 26 - 21^2 \times 5^2 \\ &= 21^2 \times 5 \times 47 \end{aligned}$$

(b) We want all 4 letter words beginning and ending with consonants that are not in $C_2 \cap C_3$, which is $21^2 \times 26^2 - 21^4$.

4.1.3 (a) If everyone who lost an eye also lost an arm, a leg and an ear, then there would be 70 people who lost all four.

(b) Let A be the set of people who lost an arm and L the set who lost a leg. How small can $A \cap L$ be? We have

$$|A \cap L| \;=\; |A| + |L| - |A \cup L| \;=\; 165 - |A \cup L| \;\geq\; 165 - 100 \;=\; 65.$$

We can now look at the set $D = A \cap L$ of double amputees and ask now many must have lost an eye. As above, we have

$$|D \cap I| \;=\; |D| + |I| - |D \cup I| \;\geq\; 65 + 70 - 100 \;=\; 35,$$

where I is the set of people who have lost an eye. Finally, we combine these people with the 75 who have lost an ear to conclude that at least $35 + 75 - 100 \;=\; 10$ must have lost all four. Thus $p \geq 10$. We can achieve this by insisting that everyone lost at least three things. If the people are numbered 1–100, we can do it as follows:

lost arm:	1–80
lost leg:	1–65 and 81–100
lost eye:	1–35 and 66–100
lost ear:	1–10 and 36–100

4.1.5 (a) A number x has a factor in common with N if and only if it is divisible by one of the primes that divide N. Thus an element of $\underline{N}$ has no factor in common with N if and only if it is in none of the sets S_k.

(b) The intersection on the left side is the set of $x \in \underline{N}$ that are multiples of $b = p_{i_1} \cdots p_{i_r}$. These are b, $2b$, $3b$ …, $(N/b)b$. Thus the set has N/b elements, as was to be proved.

(c) By (4.3) and the previous result, we have

$$\varphi(N) \;=\; \sum_{I \subseteq \underline{n}} (-1)^{|I|} \frac{N}{\prod_{i \in I} p_i} \;=\; N \sum_{I \subseteq \underline{n}} \prod_{i \in I} \left(\frac{-1}{p_i} \right).$$

Replacing x_i by $-1/p_i$ in Example 1.14, we obtain the desired result.

4.1.7. Let S_i be those lists in which c_i is adjacent to c_i. Consider a list in $S_{i_1} \cap \cdots \cap S_{i_r}$. Using the hint, this can be thought of as a list made from $2m - r$ symbols, where for the present we regard the two occurrences of the symbol c_i as different Since the list is a rearrangement of the symbols, there are $(2m - r)!$ such lists. However, $m - r$ pairs of the symbols are identical and we have treated them as different. There are 2^{m-r} ways to treat such symbols as different. Thus $N_r = \binom{m}{r}(2m-r)!/2^{m-r}$.

4.1.9. The proof is practically the same as that given for Theorem 4.1. Instead of asking how much $s \in S$ contributes to the sums, ask how much $\Pr(s)$ contributes.

4.1.11 (a) The products are 1 or 0 according as s belongs to precisely the sets S_i, $i \in K$ or not. Thus the inner sum is 1 or 0 according as s belongs to precisely k sets or not.

(b) Simply expand using the distributive law as in the previous exercise.

(c) The first part is just a rearrangement: Instead of choosing K and then J, first choose L (corresponding to $J \cup K$) and then choose K. The second part arises because there are $\binom{|L|}{k}$ ways to choose K.

(d) Move the sum over $s \in S$ inside the other sums and collect terms according to $|L|$.

4.1.13 (a) Let the notation be as in the proof of the Principle of Inclusion and Exclusion. The proof given in the text is easily adjusted to prove s contributes exactly $c_{t-1}(X)$ to $\sum_{i=0}^{t-1}(-1)^i S_i$. Thus the sum will be a lower bound when t is even and an upper bound when t is odd. Including the term $(-1)^t S_t$ in the sum changes upper bounds to lower bounds and vice versa since we are now considering $c_t(X)$. By considering the cases of t even and t odd separately, it is easy to see that the inequalities follow.

(b) This can be proved by induction on t using $\binom{|X|}{t} = \binom{|X|-1}{t} + \binom{|X|-1}{t-1}$.

4.1.15 (a) Let $m = 2$. Initially the N array contains

$$2: \quad N_2 \qquad 1: \quad N_1 \qquad 0: \quad N_0.$$

With $j = 0$, we do $i = 1$ and then $i = 0$. The N array now contains

$$2: \quad N_2 \qquad 1: \quad N_1 - N_2 \qquad 0: \quad N_0 - (N_1 - N_2).$$

With $j = 1$, we obtain

$$2: \quad N_2 \qquad 1: \quad (N_1 - N_2) - N2 \qquad 0: \quad N_0 - (N_1 - N_2).$$

Equation (4.16) gives

$$E_2 \;=\; N_2 \qquad E_1 \;=\; N_1 - 2N_2 \qquad E_0 \;=\; N_0 - N_1 + N_2,$$

which agrees with the values computed by the algorithm. You can carry out similar calculations for $m = 3$.

(b) This can be done by carefully carrying out the steps in the algorithm.

(c) After no iterations (that is, at the start of the algorithm), N_r contains s as many times as there is set of r indices for which (4.17) is true. If s appears in exactly p of the S_i, this number is $\binom{p}{r}$. We now use induction on t, having done the case $t = 0$. After $t - 1$ iterations, formula (4.18) is true when t is replaced by anything smaller in it. In particular, it holds with t replaced by $t - 1$.

We must now focus on the inner loop of the algorithm. What does it do? Since N_m never changes, neither does N_m^*. Formula (4.18) gives 0 or 1 for all t according as $p < m$ or $p = m$ ($p > m$ is impossible). This is the correct answer for both N_m and E_m.

Back to the action of the inner loop. Again we can prove it by induction, but now we are going from N_m^* down to N_0^*. We dealt with N_m^* in the previous paragraph. If the inner loop has done the correct thing with N_{r+1}^*, then the number of times s appears in the new version of N_r^* is $\mu(p, r, t-1) - \mu(p, r+1, t)$. There are various cases to consider. We'll just look at one, namely $\binom{p-(t-1)}{r-(t-1)} - \binom{p-t}{(r+1)-t}$. Using $\binom{n+1}{k} \;=\; \binom{n}{k} + \binom{n}{k-1}$, we have

$$\binom{p-(t-1)}{r-(t-1)} - \binom{p-t}{(r+1)-t} \;=\; \binom{p-t+1}{r-t+1} - \binom{p-t}{r-t+1} \;=\; \binom{p-t}{r-t},$$

which is what we needed to prove. We leave the other cases in (4.18) to you. The last sentence in the exercise follows from the fact that all the numbers we calculate are nonnegative. (This takes care of the problem of how we should interpret the multiset difference $A - B$ if s appears more often in B than it does in A.)

When $t \geq m$, the only time the binomial coefficient is used in (4.18) is when $t = p = m$ and it then has the value $\binom{0}{r-m}$, which is zero unless $r = m$, when it is 1. Thus, for $t \geq m$, $\mu(p, r, t)$ equals 1 if $r = p$ and 0 otherwise. Hence N_r^* is a set containing precisely those elements that are in exactly r of the S_i.

(d) This is implicit in the proof for (c).

4.1.17. Let S_i be the subset of $\underline{n}$ divisible by a_i. Then, N_t is the sum over all t-subsets T of A of $\lfloor n/\mathrm{lcm}(T)\rfloor$, where $\mathrm{lcm}(T)$ is the least common multiple of the elements of T and the floor $\lfloor x\rfloor$ is the largest integer not exceeding x. For the various parts of the exercise you need the following.

- If all elements of A divide n, then $\lfloor n/\mathrm{lcm}(T)\rfloor = n/\mathrm{lcm}(T)$.
- If no two elements of A have a common factor, then $\mathrm{lcm}(T) = \prod_{i\in T} i$.

(a) Using the previous comments in the special case $k = 0$, we obtain after some algebra

$$\sum_{i=0}^{m}(-1)^i N_i \;=\; n\prod_{a\in A}\left(1-\frac{1}{a}\right),$$

which is the Euler phi function when A is the set of prime divisors of n.

(b) The comment for (a) applies in this case as well.

(c) There is no simple formula even when $k = 0$ because the floor function cannot be eliminated.

(d) Now we cannot even eliminate the lcm function.

4.1.19. In all cases, what we must do is prove that (P-1), (P-2) and (P-3) hold. We omit most of them.

(d) Since $x/x = 1$, (P-1) is true. Suppose that $x\rho y$ and $y\rho x$. Then x/y and y/x are both integers. Since $(x/y)(y/x) = 1$, the only possible integer values for x/y and y/x are ± 1. Since x and y are positive, it follows that $x/y = 1$ and so (P-2) is true. Suppose that x/y and y/z are integers. Then so is x/z and so (P-3) is true.

4.1.21. Since every set is the union of itself, $x\rho x$. Suppose $x\rho y$ and $y\rho x$. Let b_y be a block of y. Since $x\rho y$, $b_x \subseteq b_y$ for some block b_x of x. Since $y\rho x$, $b'_y \subseteq b_x$ for some block b'_y of y. Since blocks of a partition are either equal or disjoint and since $b'_y \subseteq b_x \subseteq b_y$, we have $b'_y = b_y$ and so $b_x = b_y$. This proves that every block of y is a block of x. Hence $x = y$ and so (P-2) is true. It is easy to prove (P-3).

4.1.23 (a) With each element $s \in S$, associate a set $g(s)$ such that $s \in S_i$ if and only if $i \in g(s)$. Then E_k counts those $s \in S$ for which $|g(s)| = k$. Since the number of $s \in S$ with $g(s) = y$ is $e(y)$, the sum of $e(y)$ over $|y| = k$ also counts those s.

(b) An element s is counted in (4.14) if and only if it belongs to all S_i for which $i \in x$. This is the same as the definition of the set intersection.

(c) The sum of $e(x)$ over all x of since k is E_k. Putting this together with (4.15), we have

$$E_k \;=\; \sum_{|x|=k}\sum_{y\supseteq x}(-1)^{|y|-k}f(y) \;=\; \sum_{|y|\geq k}\sum_{\substack{x\subseteq y\\ |x|=k}}(-1)^{|y|-k}f(y) \;=\; \sum_{|y|\geq k}\binom{|y|}{k}(-1)^{|y|-k}f(y).$$

The sum of $f(y)$ in (b) over all y of size t is N_t. Collecting terms according to $|y|$, we have

$$E_k \;=\; \sum_{t=k}^{m}\binom{t}{k}(-1)^{t-k}N_t \;=\; \sum_{i=0}^{m-k}\binom{i+k}{k}(-1)^i N_{k+i},$$

where we set $t = i + k$. Now use $\binom{i+k}{k} \;=\; \binom{k+i}{i}$.

Section 4.2

4.2.1. The number of 6-long sequences made with B, R and W is $3^6 = 729$, which is much too long. The number of 6-long sequences in which adjacent beads differ in color is $3 \times 2^5 = 96$, which is more manageable, but still quite long. We won't list them. We could "cheat" by being a bit less mechanical: If the necklace contains a B, we could start with it. There are $2^5 = 32$ such necklaces, a manageable number. The only necklace without B must alternate R and W, so there is only one of them. Here are the 32 other necklaces, where a number preceding a necklace is the first place it appears in the list when considered circularly or flipped over. A zero means it was rejected because the first and last beads are the same.

1: BRBRBR	2: BRBRBW	0: BRBRWB	3: BRBRWR	2: BRBWBR	4: BRBWBW
0: BRBWRB	5: BRBWRW	0: BRWBRB	6: BRWBRW	0: BRWBWB	7: BRWBWR
3: BRWRBR	8: BRWRBW	0: BRWRWB	9: BRWRWR	2: BWBRBR	4: BWBRBW
0: BWBRWB	8: BWBRWR	4: BWBWBR	10: BWBWBW	0: BWBWRB	11: BWBWRW
0: BWRBRB	7: BWRBRW	0: BWRBWB	7: BWRBWR	5: BWRWBR	11: BWRWBW
0: BWRWRB	12: BWRWRW				

4.2.3 (a) Since 4 beads are used, at most 4 different kinds of beads are used. We can construct an arrangement of beads by choosing the number of types that must appear (1, 2, 3 OR 4), choosing that many types of beads from the r types AND then choosing an arrangement using all of the types of beads that we chose.

(b) Trivially, $f(1) = 1$. For $f(2)$, our decision will be the number of beads of the first type that appear. After that, it is easy. This gives us $1 + 2 + 1 = 4$. For $f(3)$, our decision will be which bead appears twice. This gives us $3 \times 2 = 6$ For $f(4)$, each bead appears once and there are 3 possibilities. Thus

$$F(r) = \binom{r}{1} + \binom{r}{2}4 + \binom{r}{3}6 + \binom{r}{4}3,$$

which can be rewritten as $r(r+1)(r^2+r^2)/8$, if desired.

4.2.5. The problem can be solved by either decision tree method. It is useful to note that all solutions must begin with h because any board that starts with v can be flipped about a NW-SE (135°) diagonal to give one that starts with h. Also note that a lexically least sequence that starts with hv determines the entire sequence. (To see this, note that it starts hvv and look at rotations of the board.)

We will use the second method. Our first decision will be the number of entire rows and/or columns that are covered by two whole dominoes. For example, two dominoes in the top row or two dominoes in the third column. Note that we cannot simultaneously cover a row and a column because they overlap. Let the number be L. The possible values of L are 0, 1, 2 and 4. (You should find it easy to see why $L = 3$ is impossible.) Note that we can always use the symmetries to make the first domino horizontal. For $L = 4$, there is obviously only one solution and its lex minimal form is *hhhhhhhh*. For $L = 0$, we use Method 1 to obtain *hvvhvvhh* as the the only solution. (Beware: reading the sequence in reverse does not correspond to a symmetry of the board.) For $L = 1$, we note that the entire row or column must be at the edge of the board. Suppose it is the first row. Refer back to Figure 3.15 to see that the only way to complete the board without increasing L is *hvvvvh*. This is already lex minimal: *hhhvvvvh*. Suppose $L = 2$. By rotation, we can assume we have two full rows and, because they cannot be in the middle, one of them is the first row. Again, refer to Figure 3.15 to find how many ways we can complete the board with one more horizontal row. This leads to six solutions: *hhhhhvvh*, *hhhvvhhh*, *hhhhvhvh*, *hhvhvhhh*, *hhhhvvvv* and *hhvvvvhh*. This gives a total of nine solutions.

4.2.7. When we write out our answers, they will be in the form suggested in the problem, without the surrounding boxes. To obtain the lex least solutions, we must linearly order the faces. Our order will be the line of four side faces from left to right, then the top and, finally, the bottom. We use B, R and W to denote the colors. and b, r and w to denote the number of faces of each color.

(a) Our first decision will be the number of black faces. By interchanging black and white, a solution with b black faces can be converted to one with $6-b$, so we only need look at $b = 0$ 1, 2 and 3. For $b = 0$ and $b = 1$, there are obviously only one solution. For $b = 2$, we must decide whether to put the second black face adjacent or opposite the first one. Here are the 4 solutions for $b < 3$.

$$\begin{array}{cccc} \text{W} & \text{W} & \text{W} & \text{W} \\ \text{WWWW} & \text{BWWW} & \text{B B WW} & \text{BWBW} \\ \text{W} & \text{W} & \text{W} & \text{W} \end{array}$$

For $b = 3$, our second decision is whether or not all three black faces share a common vertex. This leads to just 2 solutions:

$$\begin{array}{cc} \text{B} & \text{W} \\ \text{B B WW} & \text{B B BW} \\ \text{W} & \text{W} \end{array}$$

Doubling the answers for $b < 3$ to get those for $b > 3$ gives us 10 solutions.

(b) In the previous solution, we can limit ourselves to $b \le 3$. When $b = 3$, we need to check whether or not one solution is converted to the other when black and white are interchanged. They are not, so $b = 3$ still gives 2 solutions for a total of 6.

(c) The mirror image of each of the 10 solutions is equivalent to itself, so there are still 10 solutions.

(d) Our first decision will be the list b, r, w. By interchanging colors, we need only consider the situations where $b \le r \le w$. This gives us 1,1,4, 1,2,3 and 2,2,2. Interchanging colors in all possible ways gives rise to 3, 6 and 1 solutions, respectively, for each solution found. For 1,1,4, our decision will be whether B and R are on adjacent or opposite faces. Each leads to one coloring. For 1,2,3 our first decision will be the number of R's that are adjacent to the B. One adjacency gives 1 solution and two give 2 solutions, depending on whether the R's are adjacent or opposite each other. For 2,2,2, our first decision will be whether or not the B's are adjacent or opposite. Our second decision will be whether or not the R's are adjacent or opposite. Each choice leads to 1 solution except when the B's are adjacent and the R's are adjacent. In this case there are more solutions. One possibility is to have the 4 sides be BBRR. Another possibility is to have the 4 sides be BBRW and then place the additional R on either the top or the bottom. These last two possibilities are mirror images of each other, but we cannot transform one to the other with just rotations. The solutions are given in Figure S.4.1. This gives us $2 \times 3 + 3 \times 6 + 6 = 30$ solutions.

(e) If all 3 colors appear, there are 30 solutions. If only 1 color appears, there are obviously 3 solutions. What if exactly 2 colors appear, we can first choose the 2 colors AND then use them. By the first part of this exercise, there are $10 - 2 = 8$ ways to use the colors so that both appear. Thus we have $30 + 3 + \binom{3}{2}8 = 57$ solutions.

(f) Note that no color can appear more than 3 times on any given cube. Also note that at most 6 colors appear on any given cube. By looking over our previous work, we find, in the notation of Exercise 4.2.4, that $f(0) = f(1) = 0$, $f(2) = 1$ and $f(3) = 8$. By looking at decision trees for the color counts 1,1,1,3 and 1,1,2,2, we find that $f(4) = \binom{4}{1}2 + \binom{4}{2}5 = 32$. Consider $f(5)$ which has just the one color count list 1,1,1,1,2. There is one way to place the repeated colors. The

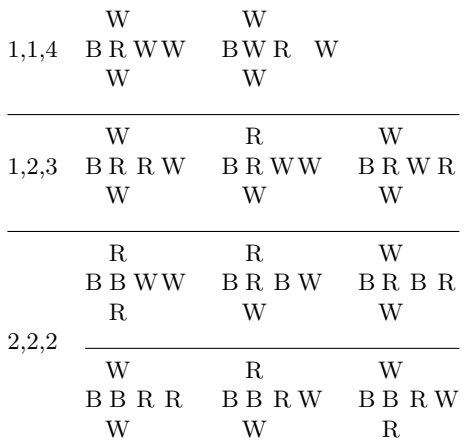


Figure S.4.1 The distinct painted cubes with various numbers of faces painted Black, Red and White.

partially colored cube can be transformed into itself be leaving it fixed or by rotating it so that the two colored faces are interchanged. This means that whenever we color the remaining 4 faces with 4 *distinct* colors, there will be exactly one other coloring that is equivalent to it. Thus $f(5) = \binom{5}{1}(4!/2) = 60$. If you experiment a bit, you will discover that there are 24 symmetries of the cube. If all the faces are colored differently, each of the symmetries leads to an equivalent coloring that looks different. Thus $f(6) = 6!/24 = 30$. Putting all this together, we have

$$F(r) = \binom{r}{2} + \binom{r}{3}8 + \binom{r}{4}32 + \binom{r}{5}60 + \binom{r}{6}30.$$

Section 4.3

4.3.1. The image of F is all k element subsets of $\underline{n}$. $F^{-1}(x)$ consists of all possible ways to arrange the elements of x in a list. Since we are able to count lists, we know that there are $k!$ such arrangements. We also know that $|A| = n!/(n-k)!$. Thus the coimage of F consists of $C(n,k)$ blocks all of size $k!$ and the union of these blocks has $n!/(n-k)!$ elements. Thus $C(n,k) = \frac{n!}{k!(n-k)!}$.

4.3.3. Note that $N(\gamma) = 0$ unless $\gamma \in P_8$ or $\gamma \in P_5$. In the former case, $N(\gamma) = \binom{8}{3} = 56$ and in the latter case, $N(\gamma) = \binom{2}{1}\binom{3}{1} = 6$. Thus there are $(56 + 4 \times 6)/16 = 5$ necklaces.

4.3.5 (a) The second line consists of the first line circularly shifted by c, an integer between 0 and $n-1$; i.e., the second line is $s_1, s_2, \ldots, s_n$, where $s_t = c + t$ if this is at most n and $c + t - n$, otherwise.

(b) In addition to the elements of the cyclic group, we have permutations whose second lines are cyclic shifts of $n, \ldots, 2, 1$.

(c) There are 0, 1 or 2 cycles of length 1 and the remaining cycles are all of length 2. If n is odd, there is always exactly one cycle of length 1. If n is even, there is never exactly one cycle of length 1. You can write down the cycles as follows. All numbers that are mentioned are understood to have an appropriate multiple of n added to (or subtracted from) them so that they lie between 1 and n inclusive. If n is odd, choose a cycle (k). The remaining cycles are

$(k-t, k+t)$ where $1 \le t < n/2$. If n is even, choose $k \le n/2$. There are two ways to proceed. First, we could have all cycles of the form $(k-t+1, k+t)$ where $1 \le t \le n/2$. Second, we could have (k), $(k+n/2)$ and all cycles of the form $(k-t, k+t)$ where $1 \le t < n/2$.

4.3.7. The proof in the text shows that the right side of the given equality is $|G| \sum_{g \in G} N(g)$. By (4.20), the left side is

$$\sum_{y \in S} |I_x| \;=\; |G| \sum_{y \in S} \frac{1}{|B_y|}.$$

The rest of the proof follows easily by adapting what was done in the text. This seems to be a shorter proof than the one in the text. Why didn't we use it? First, it's not particularly shorter; however, it is a bit cleaner. Unfortunately, it requires starting with the completely unmotivated double summation in which we have interchanged the order of the sums.

Section 5.1

5.1.1. The sum is the number of ends of edges since, if x and y are the ends of an edge, the edge contributes 1 to the value of $d(x)$ and 1 to the value of $d(y)$. Since each edge has two ends, the sum is twice the number of edges.

5.1.3. The graph with

$$\varphi \;=\; \begin{pmatrix} a & b & c & d & e & f & g & h & i & j & k \\ C & C & F & A & H & E & E & A & D & A & A \\ C & G & G & H & H & H & F & H & G & D & F \end{pmatrix}.$$

is isomorphic to Q. The correspondence between vertices is given by

$$\begin{pmatrix} A & B & C & D & E & F & G & H \\ H & A & C & E & F & D & G & B \end{pmatrix}$$

where the top row corresponds to the vertices of Q. The graph with

$$E \;=\; \{1,2,3,4,5,6,7,8,9,10,11\} \quad \text{and} \quad \varphi \;=\; \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ A & E & E & E & F & G & H & B & C & D & E \\ G & H & E & F & G & H & B & C & D & D & H \end{pmatrix}.$$

is not isomorphic to Q. One edge needs to be deleted from $P'(Q)$ and one added.

5.1.5 (a) There is no graph Q with degree sequence $(1,1,2,3,3,5)$ since the sum of the degrees is odd.

(b) There are such a graph. You should draw an example.

(c) Up to labeling, the graph is unique. Take $V = \{1, \ldots, 6\}$ and

$$E \;=\; \big\{\{1,6\},\, \{2,6\},\, \{2,4\},\, \{3,6\},\, \{3,5\},\, \{4,6\},\, \{4,5\},\, \{5,6\}\big\}$$

(d) A graph with degree sequence $(3,3,3,3)$ has $(3+3+3+3)/2 = 6$ edges and, of course 4 vertices. That is the maximum $\binom{4}{2}$ of edges that a graph with 4 vertices can have. It is easy to construct such a graph. This graph is called the *complete* graph on 4 vertices.

(f) There is no simple graph (or graph without loops or parallel edges) with degree sequence $(3,3,3,5)$.

(g) Similar arguments to the $(3,3,3,3)$ case apply to the complete graph with degree sequence $(4,4,4,4,4)$.

A	B	all	injections	surjections
L	L	b^a	$b(b-1)\cdots(b-a+1)$	$b!S(a,b)$
L	U	$\sum_{k\le b} S(a,k)$	1	$S(a,b)$
U	L	$\binom{a+b-1}{a}$	$\binom{b}{a}$	$\binom{a-1}{a-b}$
U	U	$\sum_{k\le b} p(a,k)$	1	$p(a,b)$

Figure S.5.1 Some basic enumeration problems.

Section 5.2

5.2.1. Let ν and ε be the bijections.

(a) This follows from the fact that ν and ε are bijections.

(b) This can be seen intuitively from the drawing of the unlabeled graph. If you want a more formal proof, first note that the degree of a vertex v is the number of edges e such that $v \in \varphi(e)$. Now use the fact that $v \in \varphi(e)$ is equivalent to $\nu(v) \in \varphi'(\varepsilon(e))$.

5.2.3 (a) This is exactly like the next problem with the transpose, ^{t}, replaced by inverse, $^{-1}$, everywhere.

(b) Let I be the $n \times n$ identity matrix. Since $A = IAI^t$, $A \simeq A$. Suppose that $A \simeq B$. Then $B = PAP^t$ for some nonsingular P. Multiplying on the left by P^{-1} and on the right by $(P^{-1})^t = (P^t)^{-1}$, we have

$$(P^{-1})B(P^{-1})^t \;=\; (P^{-1}P)A(P^t(P^{-1})^t) \;=\; (P^{-1}P)A(P^t(P^t)^{-1}) = A.$$

Thus $B \simeq A$. Suppose that $A \simeq B \simeq C$. Then we have nonsingular P and Q such that $B = PAP^t$ and $C = QBQ^t$. Thus $C = Q(PAP^t)Q^t = (QP)A(P^tQ^t) = (QP)A(QP)^t$. This proves transitivity.

5.2.5. Let $E \in \mathcal{P}_2(V)$ and $E' \in \mathcal{P}_2(V')$. Write $G = (V, E) \simeq (V', E') = G'$ if and only if there is a bijection $\nu\colon V \to V'$ such that $\{u, v\} \in E$ if and only if $\{\nu(u), \nu(v)\} \in V'$.

We could show that this is an equivalence relation by adapting the proof in Example 5.5. An alternative is to show how this definition leads to the equivalence relation for G and G' interpreted as graphs. We'll take this approach. In this case φ and φ' are identity maps. Define $\varepsilon(\{u, v\}) = \{\nu(u), \nu(v)\}$. By our definition in the previous paragraph, $\varepsilon\colon E \to E'$ is a bijection. Since φ and φ' are the identity, the requirement that $\varphi'(\varepsilon(e)) = \nu(\varphi(e))$ in the definition of graph isomorphism is satisfied.

5.2.7. The table is shown in Figure S.5.1. The entries which are 1 follow when you realize what is being counted. The LL row corresponds to ordered samples and the UL row to unordered samples, which have been considered in Chapter 1. The UL-surjection entry comes from the realization that our sample allows repetition but must include every element in b so that we are only free to choose $a - b$ additional elements. In the LU row, the fact that the range is unlabeled means that we can only distinguish functions that have different coimages. The UU row is associated with partitions of numbers. We use $p(n, k)$ to denote the number of partitions of n having exactly k parts.

Section 5.3

5.3.1. Since $E \subseteq \mathcal{P}_2(V)$, we have a simple graph. Regardless of whether you are in set C or S, following an edge takes you into the other set. Thus, following a path with an odd number of edges takes you to the opposite set from where you started while a path with an even number of edges takes you back to your starting set. Since a cycle returns to its starting vertex, it obviously returns to its starting set.

5.3.3 (a) Let $e = \{u, v\}$ and let $f = \{v, w\}$ be the other edge. Since G is simple, $u \neq w$. Since e is a cut edge, u and v are in separate components of $(V, E - \{e\})$. Thus so are u and w. Since the graph induced by $V - \{v\}$ is a subgraph of $(V, E - \{e\})$, u and w are in separate components of it as well.

(b) Take two triangles and identify their tops. The merged top is a cut vertex but the graph has no isthmus.

(c) We will prove that e is a cut edge if and only if its ends u and v, say, lie in different components of $G' = (V, E - \{e\})$. The result will then follow because, first, if C is a cycle containing e, removal of e does not leave its ends in different components, and, second, if u and v are in the same components of G', then there is a path P connecting them in G' and P and e form a cycle in G.

Now back to the original claim. If u and v are in different components of G', then e is a cut edge. Suppose e is a cut edge of G. Since G is connected and every path in G that is not a path in G' contains e, it follows that if x and y are in different components of G' any path connecting them in G contains e. Let P be such a path and let u be the end of e first reached on P when starting from x. It follows that x and u are in one component of G' and that y and v (the other end of e) are one component, too. Since x and y are in different components, so are u and v.

(d) We claim that $v \in V$ is a cut vertex of G if and only if there are two edges e and e' both containing v such that no cycle of G contains both e and e'.

Proof. Suppose that v is a cut vertex. Let x and y belong to different components of the graph G'' induced by $V - \{v\}$. Any path from x to y in G must include v. Let P be such a path and let e and e' be the two edges in P that contain v. If e and e' were on a cycle C in G, then we could remove e and e' from P and add on $C - \{e, e'\}$ to obtain a route from x to y that does not go through v. Since this contradicts the fact that x and y are in different components of G", it follows that e and e' do not lie in a cycle.

The steps can be reversed to prove that if e and e' are edges incident with v that do not lie on a cycle, then v is a cut vertex: Let x and y be the other vertices on e and e'. Since e and e' do not lie on a cycle, every path from x to y must include either e or e' (or both), and hence includes v. Since there is no path from x to y not including v, they are in different components of G".

5.3.5 (a) The graph is not Eulerian. The longest trail has 5 edges, the longest circuit has 4 edges.

(b) The longest trail has 9 edges, the longest circuit has 8 edges.

(c) The longest trail has 13 edges (an Eulerian trail starting at C and ending at D). The longest circuit has 12 edges.

(d) This graph has an Eulerian circuit (12 edges).

Section 5.4

5.4.1. We first prove that (b) and (c) are equivalent. We do this by showing that the negation of (b) and the negation of (c) are equivalent. Suppose $u \neq v$ are on a cycle of G. By Theorem 5.3, there are two paths from u to v. Conversely, suppose there are two paths from u to v. Call them $u = x_0, x_1, \ldots, x_k = v$ and $u = y_0, y_1, \ldots, y_m = v$. Let i be the smallest index such that $x_i \neq y_i$. We may assume that $i = 1$ for, if not, redefine $u = x_{i-1}$. On the new paths, let $x_a = y_b$ be the smallest $a > 0$ for which some x_j is on the y path. The walk

$$u = x_0,\ x_1, \ldots, x_a = y_b,\ y_{b-1}, \ldots, y_0 = u$$

has no repeated vertices except the first and last and so is a cycle. (A picture may help you visualize what is going on. Draw the x path intersecting the y path several times.)

We now prove that (d) implies (b). Suppose that G has a cycle, $v_0, v_1, \ldots, v_k, v_0$. Remove the edge $\{v_0, v_k\}$. In any walk that uses that edge, replace it with the path $v_0, v_1, \ldots, v_k$ or its reverse, as appropriate. Thus the graph is still connected and so the edge $\{v_0, v_k\}$ contradicts (d).

5.4.3 (a) By Exercise 5.1.1, we have $\sum_{v \in V} d(v) = 2|E|$. By 5.4(e), $|E| = |V| - 1$. Since

$$2|V| \ = \ \sum_{v \in V} 2, \quad \text{we have} \quad 2 \ = \ 2|V| - 2|E| \ = \ \sum_{v \in V} (2 - d(v)).$$

(b) We give three solutions. The first uses the previous result. The second uses the fact that each tree except the single vertex has at least two leaves. The third uses the fact that trees have no cycles.

Suppose that T is more than just a single vertex. Since T is connected, $d(v) \neq 0$ for all v. Let n_k be the number of vertices of T of degree k. By the previous result, $\sum_{k \geq 1} (2 - k) n_k = 2$. Rearranging gives $n_1 = 2 + \sum_{k \geq 2} (k - 2) n_k$. If $n_m \geq 1$, the sum is at least $m - 2$.

For the second solution, remove the vertex of degree m to obtain m separate trees. Each tree is either a single vertex, which is a leaf of the original tree, or has at least two leaves, one of which must be a leaf of the original tree.

For the third solution, let v be the vertex of degree m and let $\{v, x_i\}$ be the edges containing v. Each path starting v, x_i must eventually reach a leaf since there are no cycles. Call the leaf y_i. These leaves are distinct since, if $y_i = y_j$, the walk $v, x_i, \ldots, y_i = y_j, \ldots, x_j, v$ would lead to a cycle.

(c) Let the vertices be u and v_i for $1 \leq i \leq m$. Let the edges be $\{u, v_i\}$ for $1 \leq i \leq m$.

(d) Let $N = n_3 + n_4 + \cdots$, the number of vertices of degree 3 or greater. Note that $k - 2 \geq 1$ for $k \geq 3$. By our earlier formula, $n_1 \geq 2 + N$. If $n_2 = 0$, $N = |V| - n_1$ and so we have $n_1 \geq 2 + |V| - n_1$. Thus $n_1 \geq 1 + |V|/2$. Similarly, if $n_2 = 1$, $N = |V| - n_1 - 1$ and, with a bit of algebra, $n_1 \geq (1 + |V|)/2$.

(e) A careful analysis of the previous argument shows that the number of leaves will be closest to $|V|/2$ if we avoid vertices with high degrees. Thus we will try to make our vertices of degree three or less. We will construct some RP-trees, T_k with k leaves. Let T_1 the isolated vertex. For $k > 1$, let T_k have two children, one a single vertex and the other the root of T_{k-1}. Clearly T_k has one more leaf and one more nonleaf than T_{k-1}. Thus the difference between the number of leaves and nonleaves is the same for all T_k. For T_1 it is one.

5.4.5. Since the tree has at least 3 vertices, it has at least $3 - 1 = 2$ edges. Let $e = \{u, v\}$ be an edge. Since there is another edge and a tree is connected, at least one of u and v must lie on another edge besides e. Suppose that u does. It is fairly easy to see that u is a cut vertex and that e is a cut edge.

5.4.7 (a) The idea is that for a rooted planar tree of height h, having at most 2 children for each non-leaf, the tree with the most leaves occurs when each non-leaf vertex has exactly 2 children. You should sketch some cases and make sure you understand this point. For this case $l = 2^h$ and so $\log_2(l) = h$. Any other rooted planar tree of height h, having most 2 children for each non-leaf, is a subtree (with the same root) of this maximal-leaf binary tree and thus has fewer leaves.

(b) The height h can be arbitrarily large.

(c) $h = l - 1$.

(d) $\lceil \log_2(l) \rceil$ is a lower bound for the height of *any* binary tree with l leaves. It is easy to see that you can construct a full binary tree with l leaves and height $\lceil \log_2(l) \rceil$.

(e) $\lceil \log_2(l) \rceil$ is the minimal height of a binary tree.

5.4.9 (a) A binary tree with 35 leaves and height 100 is possible.

(b) A full binary tree with 21 leaves can have height at most 20. So such a tree of height 21 is impossible.

(c) A binary tree of height 5 can have at most 32 leaves. So one with 33 leaves is impossible.

(d) A full binary tree with 65 leaves has minimal height $\lceil \log_2(65) \rceil = 7$. Thus a full binary tree with 65 leaves and height 6 is impossible.

5.4.11 (a) Breadth-first: $MIAJKCEHLBFGD$,
Depth-first: $MICIEIHFHGHDHIMAMJMKLKBKM$,
Pre-order: $MICEHFGDAJKLB$,
Post-order: $CEFGDHIAJLBKM$.

(b) The tree is the same as in part (a), reflected about the vertical axis, with vertices A and J removed.

(c) It is not possible to reconstruct a rooted plane tree given just its pre-order vertex list. A counterexample can be found using just three vertices.

(d) It is possible to reconstruct a rooted plane tree given its pre-order and post-order vertex list. If the root is X and the first child of the root is Y, it is possible to reconstruct the pre-order and post-order vertex lists of the subtree rooted at Y from the pre-order and post-order vertex lists of the tree. In the same manner, you can reconstruct the pre-order and post-order vertex lists of the subtrees rooted at the other children of the root X. Now do the same trick on these subtrees. Try this approach on an example.

Section 5.5

5.5.1. Let D be the domain suggested in the hint and define $f\colon D \to \mathcal{P}_2(V)$ by $f((x,y)) = \{x,y\}$. Let $G(D) = (V,\psi)$ where $\psi(e) = f(\varphi(e))$.

5.5.3. Let $V = \{u,v\}$ and $E = \{(u,v),(v,u)\}$.

5.5.5. You can use the notation and proof of Example 5.5 provided you change all references to two element sets to references to ordered pairs. This means replacing $\{x,y\}$ with (x,y), $\{\nu(x),\nu(y)\}$ with $(\nu(x),\nu(y))$ and $\mathcal{P}_2(V_i)$ with $V_i \times V_i$.

5.5.7. "The statements are all equivalent" means that, given any two statements v and w, we have a proof that v implies w. Suppose D is strongly connected. Then there is a directed path $v = v_1, v_2, \ldots, v_k = w$. That means we have proved v_1 implies v_2, that v_2 implies v_3 and so on. Hence v_1 implies v_k.

5.5.9. Let $e = (u_1,u_2)$. For $i = 2,3,\ldots$, as long as $u_i \neq u_1$ choose an edge (u_i,u_{i+1}) that has not be used so far. It is not hard to see that $d_{\text{in}}(u_i) = d_{\text{out}}(u_i)$ implies this can be done. In this way we obtain a directed trail starting and ending at u_1. This may not be a cycle, but a cycle containing e can be extracted from it by deleting some edges.

5.5.11 (a) It's easy to see this pictorially: Suppose there were an isthmus $e = \{u,v\}$. Then G consists of the edge e, a graph G_1 containing u, and another graph G_2 containing v. Suppose e is directed as (u,v). Clearly one can get from G_1 to G_2 but one cannot get back along directed edges, contradicting strongly connectedness.

Here is a more formal proof. Suppose there were such a path, say $v = v_1, v_2, \ldots, v_k = u$ It does not contain the directed edge e (since e goes in the wrong direction). Now look at the original undirected graph. We claim removal of $\{u,v\}$ does not disconnect it. The only problem would be a path that used $\{u,v\}$ to get from, say x to y, say $x,\ldots,x',u,v,y',\ldots,y$. The walk $x,\ldots,x',v_k,\ldots,v_2,v_1,y',\ldots,y$ connects u and v without using the edge $\{u,v\}$.

(b) See Exercise 6.3.14 (p. 170).

5.5.13 (a) For all $x \in S$, $x|x$. For all $x,y \in S$, if $x|y$ and $x \neq y$, then y does not divide x. For all $x,y,z \in S$, $x|y$, $y|z$ implies that $x|z$.

(b) The covering relation

$$H = \{(2,4),(2,6),(2,10),(2,14),(3,6),(3,9),(3,15),(4,8),(4,12),(5,10),(5,15),(6,12),(7,14)\}.$$

5.5.15 (a) There are n^{n-2} trees. Since a tree with n vertices has $n-1$ edges, the answer is zero if $q \neq n-1$. If $q = n-1$, there are $\binom{\binom{n}{2}}{n-1}$ graphs. Thus the answer is $n^{n-2}\binom{\binom{n}{2}}{n-1}^{-1}$ when $q = n-1$.

(b) We have

$$\binom{\binom{n}{2}}{n-1} < \frac{\binom{n}{2}^{n-1}}{(n-1)!} = \frac{n^{n-1}(n-1)^{n-1}}{2^{n-1}\,(n-1)!} < \frac{n^{n-1}}{2^{n-1}/e^{n-1}} = \left(\frac{ne}{2}\right)^{n-1}.$$

Using this in the answer to (a) gives the result we want. It turns out that

$$n^{n-2}\binom{\binom{n}{2}}{n-1}^{-1} \sim \sqrt{\pi/2n}\,(2/e)^n,$$

which differs from our estimate by a constant times $n^{1/2}$.

Section 5.6

5.6.1. Let A and B be the partition of the vertices guaranteed by the definition of a bipartite graph. Let $k = |A|$, number the vertices in A with 1 to k and those is B with $k+1$ to n. Since no edges connect vertices in A to each other, $A(G)$ has a $k \times k$ block of zeroes in its upper left corner. Similarly B gives a block in the lower right corner.

5.6.3 (a) $a_{i,j}^{(k)}$ is the sum over all $t_1, \ldots, t_{k-1}$ of $a_{i,t_1} a_{t_1,t_2} \cdots a_{t_{k-1},j}$. Each of these products is 0 or 1, so the sum is nonzero if and only if some product is nonzero. This happens if and only if each factor in the product is nonzero. This happens if and only if the vertices $i, t_1, \ldots, t_{k-1}, j$ form a walk.

(b) We can construct a path from a walk by jumping over pieces that form cycles. Thus the shortest walk from i to j is a path. Here's a more formal argument. Suppose that $W = (i, t, \ldots, v, j)$ is the shortest walk from i to j. If it is not a path, then there must be repeated vertices in the list. Let u be such a vertex. Remove all vertices from the sequence after the first occurrence of u up to and including the last occurrence of u. The result is a shorter walk, contradicting the minimality of W.

(c) The obvious idea is to repeat the previous statement with $i = j$: "The shortest walk from i to i is a cycle." This is not true. If $\{i, j\}$ is an edge, then i, j, i is the shortest walk from i to j but it is not a cycle. The result would be true if we were looking at oriented simple graphs because an edge can be traversed in only one direction. All we can claim is that any odd length walk from i to i contains a cycle.

We can modify the situation a bit by looking at an edge $\{i, j\}$ of the graph. Let H be the graph obtained by removing it; i.e., by setting $a_{i,j} = a_{j,i} = 0$. The shortest walk from j to i in H together with the edge $\{i, j\}$ is a cycle of G. This follows from the previous result and the definitions of path and cycle.

(d) Following the hint, $B^k = \sum_{t=0}^{k} \binom{k}{t} B^t$ by the binomial theorem. Since $\binom{k}{t} > 0$, $b_{i,j}^{(k)}$ is nonzero if and only if $a_{i,j}^{(t)} \neq 0$ for some t with $0 \leq t \leq k$. $t = 0$ gives the identity matrix, so $b_{i,i}^{(k)} \neq 0$ for all k. For $i \neq j$, $b_{i,j}^{(k)} \neq 0$ if and only if there is a walk from i to j for some $t \leq k$, and thus if and only if there is a path for some $t \leq k$. Since paths of length t contain $t+1$ distinct vertices, no path is longer than $n-1$. Thus there is a path from i to $j \neq i$ if and only if $b_{i,j}^{(k)} \neq 0$ for all $k \geq n-1$.

5.6.5. We claim that $A(D)$ is nilpotent if and only if there is no vertex i such that there is a walk from i to i (except the trivial walk consisting of just i).

First suppose that there is a nontrivial walk from i to i containing k edges. Let $C = A(D)^k$. It follows that all entries of C are nonnegative and $c_{i,i} \neq 0$. Thus $c_{i,i}^{(m)} \neq 0$ for all $m > 0$. Hence $A(D)$ is not nilpotent.

Conversely, suppose that $A(D)$ is not nilpotent. Let n be the number of vertices in D and suppose that i and j are such that $a_{i,j}^{(n)} \neq 0$, which we can do since $A(D)$ is not nilpotent. There must be a walk $i = v_0, v_1, v_2, \ldots, v_n = j$. Since this sequence contains $n+1$ vertices, there must be a repeated vertex. Suppose that $k < l$ and $v_k = v_l$. The sequence $v_k, v_{k+1}, \ldots, v_l$ is a nontrivial walk from v_k to itself.

Section 6.1

6.1.1 (a) One description of a tree is: a connected graph such that removal of any edge disconnects the tree. Since an edge connects only two vertices, we will obtain only two components by removing it.

(b) Note that T with e removed and f added is a spanning tree. Since T has minimum weight, the result follows.

(c) The graph must have a cycle containing e. Since one end of e is in T_1 and the other in T_2, the cycle must contain another connector besides e.

(d) Since T^* with e removed and f added is a spanning tree, the algorithm would have removed f instead of e if $\lambda(f) > \lambda(e)$.

(e) By (b) and (d), $\lambda(f) = \lambda(e)$. Since adding f connects T_1 and T_2, the result is a spanning tree.

(f) Suppose T^* is not a minimum weight spanning tree. Let T be a minimum weight spanning tree so that the event in (a) occurs as late as possible. It was proven in (e) that we can replace T with another minimum weight spanning tree such that the disagreement between T and T^*, if any, occurs later in the algorithm. This contradicts the definition of T.

6.1.3 (b) Let Q_1 and Q_2 be two bicomponents of G, let v_1 be a vertex of Q_1, and let v_2 be a vertex of Q_2. Since G is connected, there is a path in G from v_1 to v_2, say $x_1, \ldots, x_p$. You should convince yourself that the following pseudocode constructs a walk $w_1, w_2, \ldots$ in $\mathcal{B}(G)$ from Q_1 to Q_2.

```
Set w_1 = Q_1, j=2, and k = 0.
While there is an x_i ∈ P(G) with i > k.
   Let i > k be the least i for which x_i ∈ P(G).
   If i = p
      Set Q = Q_2.
   Else
      Let Q be the bicomponent containing {x_i, x_{i+1}}.
   End if
   Set w_j = x_i, w_{j+1} = Q, k = i, and j = j + 2.
End while
```

(c) Suppose there is a cycle in $\mathcal{B}(G)$, say $v_1, Q_1, \ldots, v_k, Q_k, v_1$, where the Q_i are distinct bicomponents and the v_i are distinct vertices. Set $v_{k+1} = v_1$. By the definitions, there is a path in Q_i from v_i to v_{i+1}. Replace each Q_i in the previous cycle with these paths after removing the endpoints v_i and v_{i+1} from the paths. The result is a cycle in G. Since this is a cycle, all vertices on it lie in the same bicomponent, which is a contradiction since the original cycle contained more than one Q_i.

(d) Let v be an articulation point of the simple graph G. By definition, there are vertices x and y such that every path from x to y contains v. From this one can prove that there are edges $e = \{v, x'\}$ and $f = \{v, y'\}$ such that every path from x' to y' contains v. It follows that e and f are in different bicomponents. Thus v lies in more than one bicomponent.

Suppose that v lies in two bicomponents. There are edges $e = \{v, w\}$ and $f = \{v, z\}$ such that $e \not\sim f$. It follows that every path from w to z contains v and so v is an articulation point.

6.1.5 (a) Since there are no cycles, each component must be a tree. If a component has n_i vertices, then it has $n_i - 1$ edges since it is a tree. Since $\sum n_i$ over all components is n and $\sum(n_i - 1)$ over all components is k, $n - k$ is the number of components.

(b) By the previous part, H_{k+1} has one less component than G_k does. Thus at least one component C of H_{k+1} has vertices from two or more components of G_k. By the connectivity of C, there must be an edge e of C that joins vertices from different components of G_k. If this edge is added to G_k, no cycles arise.

(c) By the definition of the algorithm, it is clear that $\lambda(g_1) \leq \lambda(e_1)$. Suppose that $\lambda(g_i) \leq \lambda(e_i)$ for $1 \leq i \leq k$. By the previous part, there is some e_j with $1 \leq j \leq k+1$ such that G_k together with e_j has no cycles. By the definition of the algorithm, it follows that $\lambda(g_{k+1}) \leq \lambda(e_j)$. Since $\lambda(e_j) \leq \lambda(e_{k+1})$ by the definition of the e_i's, we are done.

6.1.7 (a) Hint: For (1) there are four spanning trees. For (2) there are 8 spanning trees. For (3) there are 16 spanning trees.

(b) Hint: For (1) there is one. For (2) there are two. For (3) there are two.

(c) Hint: For (1) there are two. For (2) there are four. For (3) there are 6.

(d) Hint: For (1) there are two. For (2) there are three. For (3) there are 6.

6.1.9 (a) Hint: There are 21 vertices, so the minimal spanning tree has 20 edges. Its weight is 30.

(b) Hint: Its weight is 30..

(c) Hint: Its weight is 30.

(d) Hint: Note that K is a the only vertex in common to the two bicomponents of this graph. Whenever this happens (two bicomponents, common vertex), the depth-first spanning tree rooted at that common vertex has exactly two "principal subtrees" at the root. In other words, the root of the depth-first spanning tree has degree two. Finding depth first spanning trees of minimal weight is, in general, difficult. You might try it on this example.

Section 6.2

6.2.1. This is just a matter of a little algebra.

6.2.3 (a) To color G, first color the vertices of H AND then color the vertices of K. By the Rule of Product, $P_G(x) = P_H(x)P_K(x)$.

(b) Let v be the common vertex. There is an obvious bijection between pairs of colorings (λ_H, λ_K) of H and K with $\lambda_H(v) = \lambda_K(v)$ and colorings of G. We claim the number of such pairs is $P_H(x)\big(P_K(x)/x\big)$. To see this, note that, in the colorings of K counted by $P_K(x)$, each of the x ways to color v occurs equally often and so $1/x$ of the colorings will have $\lambda_K(v)$ equal to the color given by $\lambda_H(v)$.

(c) The answer is $P_H(x)P_K(x)(x-1)/x$. We can prove this directly, but we can also use (b) and (6.4) as follows. Let $e = \{v, w\}$. By the construction of G, $P_{G-e}(x) = P_H(x)P_K(x)$. By (b), $P_{G_e}(x) = P_H(x)P_K(x)/x$. Now apply (6.4).

6.2.5. Let the solution be $P_n(x)$. Clearly $P_1(x) = x(x-1)$, so we may suppose that $n \geq 2$. Apply deletion and contraction to the edge $\{(1,1),(1,2)\}$. Deletion gives a ladder with two ends sticking out and so its chromatic polynomial is $(x-1)^2P_{n-1}(x)$. Contraction gives a ladder with the contracted vertex joined to two adjacent vertices. Once the ladder is colored, there are $x-2$ ways to color the contracted vertex. Thus we have

$$P_n(x) \;=\; (x-1)^2P_{n-1}(x) - (x-2)P_{n-1}(x) \;=\; (x^2-3x+3)P_{n-1}(x).$$

The value for $P_n(x)$ now follows easily.

6.2.7. The answer is

$$x^8 - 12x^7 + 66x^6 - 214x^5 + 441x^4 - 572x^3 + 423x^2 - 133x.$$

There seems to be no really easy way to derive this. Here's one approach which makes use of Exercise 6.2.3 and $P_{Z_n}(x)$ for $n = 3, 4, 5$. Label the vertices reading around one face with a, b, c, d and around the opposite face with A, B, C, D so that $\{a, A\}$ is an edge, etc. If the edge $\{a, A\}$ is contracted, call the new vertex α. Introduce β, γ and δ similarly.

Let $e_1 = \{a, A\}$ and $e_2 = \{b, B\}$. Note that $G - e_1 - e_2$ consists of three squares joined by common edges and that $H = G_{e_1} - e_2$ is equivalent to $(G - e_1)_{e_2}$. We do H in the next paragraph. In $K = G_{e_1 e_2}$, let $f = \{\alpha, \beta\}$. $K - f$ is two triangles and a square joined by common edges and K_f is a square an a vertex v joined to the vertices of the square. By first coloring v and then the square, we see that $P_{K_f}(x) = xP_{Z_4}(x - 1)$.

Let $f_1 = \{c, C\}$, $f_2 = \{d, D\}$ and $f_3 = \{\beta, \gamma\}$. Then

- $H - f_1 - f_2$ is two Z_5's sharing β;
- $(H - f_1)_{f_2}$ is easy to do if you consider two cases depending on whether β and δ have the same or different colors, giving $x(x - 1)(x - 2)^4 + x(x - 1)^4$;
- $H_{f_1} - f_3$ is a Z_5 and a triangle with a common edge and
- $H_{f_1 f_3}$ are three triangles joined by common edges.

6.2.9. This can be done by induction on the number of edges. The starting situation involves some number n of vertices with no edges. Since the chromatic polynomial is x^n, the result is proved for the starting condition.

Now for the induction. Deletion does not change the number of vertices, but reduces the number of edges. By induction, it gives a polynomial for which the coefficient of x^k is a nonnegative multiple of $(-1)^{n-k}$. Contraction decreases both the number of vertices and the number of edges by 1 and so gives a polynomial for which the coefficient of x^k is a nonnegative multiple of $(-1)^{n-1-k}$. Subtracting the two polynomials gives one where the coefficient of x^k is an nonnegative multiple of $(-1)^{n-k}$.

Section 6.3

6.3.1. Every face must contain at least four edges and each side of an edge contributes to a face. Thus $4f \geq$ (edge sides) $= 2e$. From Euler's relation,

$$2 \;=\; v - e + f \;\geq\; v - e + e/2 \;=\; (2v - e)/2$$

and so $e \geq 2v - 4$.

6.3.3 (a) We have $2e = fd_f$ and $2e = vd_v$. Use this to eliminate v and f in Euler's relation.

(b) They are cycles.

(c) If $d_f \geq 4$ and $d_v \geq 4$, we would have $0 < 2/d_f + 2/d_v - 1 \leq 0$, a contradiction. Thus at least one of d_v and d_f is 3. Since $d_v \geq 3$, we have $2/d_v \leq 2/3$. Thus

$$0 \;<\; \frac{2}{d_f} + \frac{2}{d_v} - 1 \;\leq\; \frac{2}{d_f} - \frac{1}{3}$$

and so $d_f < 2/(1/3) = 6$. Since d_f is an integer, $d_f \leq 5$. Since $d_f \geq 3$ for a simple graph, interchanging f and v in the above gives us $d_v \leq 5$.

(d) Altogether there are 5 possibilities for the pair (d_v, d_f) by the previous part of the exercise. Given d_f and d_v, we can solve (6.9) for e. Then $vd_v = 2e$ and $fd_f = 2e$ give v and f. The five graphs turn out to be the Platonic solids with the interiors removed. (They are the tetrahedron, cube, octahedron, dodecahedron and icosahedron.)

6.3.5. The value of c is zero. Suppose when we cut as directed we cut through k edges. Each of these edges now becomes two, giving us k new edges. The same happens with the k faces. On each of the circles that we fill in with, we also get k edges and k vertices. The two circles give us 2 new faces. In summary, if we originally had $|V|$ vertices, $|E|$ edges and f faces on the torus, we now have a graph embedded on the sphere with $V|+2k$ vertices, $|E|+k+2k$ edges, and $f+k+2$ faces. From Euler's relation on the sphere,

$$2 \;=\; (|V|+2k)-(|E|+3k)+(f+k+2) \;=\; |V|-|E|+f.$$

Thus $|V|-|E|+f \;=\; 0$.

There's a subtle issue here: We described the cut as if each edge and face it encountered was different. This may not be the case, an edge (and face) can twist around the torus so that the cut meets it more than once; however, the counts are still correct. One way to see this is to imagine what happens if we cut around the face and stretch it flat. Stretching will distort our "bracelet cut" into some sort of curve that may cut through the face several times. Every time it passes through the face it creates another face, two edges and two vertices.

6.3.7. One method is to list all the simple planar graphs with $V = \underline{5}$ and find the least colorings for them. We use a theoretical argument instead.

The lex least proper coloring of $\underline{k} \subseteq V$ uses at most the first k colors. If it uses all k colors, then vertex k must be connected to each of the other vertices and the first $k-1$ vertices must use all of the first $k-1$ colors.

Let's apply these observations with $k = 5$, 4, 3 and 2 to a graph whose lex least coloring takes 5 colors. With $k = 5$, we see that vertex 5 is connected to each of the first 4 vertices and they use 4 different colors. Now, with $k = 4$, we see that vertex 4 is connected to each of the first 3 vertices and they use 3 different colors. Doing the same thing with $k = 3$ and $k = 2$, we finally see that every vertex is connected to every other; i.e., the graph is K_5, which is not planar.

6.3.9. The argument for degree 4 is correct. For degree 5, we can assume, perhaps after rotating and or flipping the graph, that $y_1, \ldots, y_5$ are assigned colors c_1, c_2, c_3, c_4 and c_2, respectively. Suppose we look at y_1 and y_3 as in the text. The argument given there is okay if we get y_1 and y_3 in separate components. If they are in the same component, we end up switching colors c_2 and c_4 in the component of the subgraph colored by c_2 and c_4 that contains v_4. The colors of $y_1, \ldots, y_4$ are now c_1, c_2, c_3 and c_2. If y_5 was not in the same component with y_4, it is colored c_2 and we are done. Unfortunately, if y_4 and y_5 are in the same component, *its color is switched to* c_4. You should convince yourself that there is no way to arrange things to avoid this possibility.

6.3.11 (a) We start with $\lambda = \left(\begin{smallmatrix}1&2&3&4&5&6&7\\1&&&&&&7\end{smallmatrix}\right)$. A cycle is $(1,2,3,4,7)$, so we now have $\lambda = \left(\begin{smallmatrix}1&2&3&4&5&6&7\\1&2&3&4&&&7\end{smallmatrix}\right)$. Another cycle is $(1,2,5,6,7)$, so we look at the path $2,5,6,7$ and choose $\lambda = \left(\begin{smallmatrix}1&2&3&4&5&6&7\\1&2&3&4&a&b&7\end{smallmatrix}\right)$ where $2 < a < b < 7$ and λ is an injection. Depending on the choice of a and b compared to 3 and 4, we have get five different 1, 7-labelings. There could be others.

(b) Any 1, 7-labeling can be converted to a 7, 1-labeling simply by defining $\lambda_{7,1}(x) = 8 - \lambda_{1,7}(x)$.

6.3.13. We'll find all s,t-labelings. Suppose $K_{3,3}$ consists of all possible edges between $1,2,3$ and a,b,c. By symmetry, we may assume that $\lambda(1) = 1$ and either $\lambda(2) = 6$ or $\lambda(a) = 6$. In the former case, condition (c) requires that $\lambda(3)$ be less than 5 and more than 2. Up to symmetry, this gives us two answers:

$$\lambda \;=\; \left(\begin{smallmatrix}1&2&3&a&b&c\\1&6&3&2&4&5\end{smallmatrix}\right) \qquad \lambda \;=\; \left(\begin{smallmatrix}1&2&3&a&b&c\\1&6&4&2&3&5\end{smallmatrix}\right).$$

Now suppose $\lambda(a) = 6$. In this case, one of $\lambda(2)$ and $\lambda(3)$ must be greater than $\lambda(b)$ and $\lambda(c)$. Thus, up to symmetry, we have $\lambda(2) = 5$. Similarly $\lambda(b) = 2$. This leads to two more answers:

$$\lambda \;=\; \left(\begin{smallmatrix}1&2&3&a&b&c\\1&5&3&6&2&5\end{smallmatrix}\right) \qquad \lambda \;=\; \left(\begin{smallmatrix}1&2&3&a&b&c\\1&5&4&6&2&5\end{smallmatrix}\right).$$

6.3.15. We know from the text that a biconnected graph has an st-labeling. If $|V| = 2$, the result is trivial. Suppose that we have an st-labeling and that $\{x, y\}$ is an edge different from $\{s, t\}$. We may assume that $\lambda(x) < \lambda(y)$. By (iii) in the definition of st-labeling, we can find a sequence $y = w_1, w_2, \ldots = t$ such that $\lambda(w_i)$ is strictly increasing and such that $\{w_i, w_{i+1}\} \in E$. Similarly, we can find $x = u_1, u_2, \ldots = s$. These two paths with $\{s, t\}$ and $\{x, y\}$ form a cycle of G and so $\{x, y\}$ and $\{s, t\}$ are in the same bicomponent.

Section 6.4

6.4.1 (a) The value of a maximum flow is 45. Every maximum flow f will have $f(q, f) = 10$. Some other values of f are also determined uniquely, but many are not; for example, the flow into r can have any value from 15 to 20. Of course, the flows on the minimum cut set are unique. There are four minimum cut sets. The one found using $\mathcal{A}(f)$ is

$$\{\{r, h\}, \{f, a\}, \{k, e\}, \{y, u\}, \{z, u\}\}.$$

The others are obtained

(i) by deleting $\{r, h\}$ and adding $\{h, a\}$ and $\{h, c\}$,

(ii) by deleting $\{y, u\}$ and $\{z, u\}$ and adding $\{u, n\}$, or

(iii) by doing both (i) and (ii).

(b) See the previous solution.

(c) The value of a maximum flow is 25. Every maximum flow f will have $f(v, q) = 10$. Some other values of f are also determined uniquely, but many are not. There is just one minimum cut set:

$$\{\{c, d\}, \{k, e\}, \{r, x\}, \{w, x\}\}.$$

(d) See the previous solution. Since we do not have tools for finding all minimum cut sets, you may not have been able to prove that the minimum cut set was unique.

6.4.3. Since no complete augmentable path exists, $\mathcal{D}_{\text{in}} \subseteq A \subseteq V - \mathcal{D}_{\text{out}}$. Since $b(v) = 0$ for $v \notin \mathcal{D}$, it follows that $\sum_{v \in A} b(v) = \sum_{v \in \mathcal{D}_{\text{in}}} b(v)$, which is the definition of the value of a flow. Recall that $b(v)$ is the sum of all flows out of v minus the sum of all flows into v. It follows that for $e = (x, y) \in E$, $b(x)$ has a contribution of $f(x, y)$ and $b(y)$ has a contribution of $-f(e)$. We distinguish four cases according as x and y are in A or B and ask what $f(e)$ contributes to $\sum_{v \in A} b(v)$.

(i) $x \in B$, $y \in B$: Then $f(e)$ contributes nothing to the sum.

(ii) $x \in A$, $y \in A$: Then $f(e)$ contributes both $f(e)$ and $-f(e)$, which gives a net contribution of zero.

(iii) $x \in A$, $y \in B$; i.e., $(e) \in \text{FROM}(A, B)$: Then $f(e)$ contributes $f(e)$ to the sum.

(iv) $x \in B$, $y \in A$; i.e., $(e) \in \text{FROM}(B, A)$: Then $f(e)$ contributes $-f(e)$ to the sum.

6.4.5 (a) Without examining the network in detail, we would need to let c_1' and c_2' (resp. c_3' and c_4') be the sum of the capacities of edges leaving (resp. entering) the corresponding P_i'. That way we can guarantee the capability of supplying (resp. removing) as much fluid as the pump could possibly send out to (resp. get in from) other other sources. If we know all the maximum flows for the original network, we may be able to improve on this: We need to set c_i' to the largest net flow out of (resp. into) D_i for all maximum flows in the original network. This leads to no improvement in this case.

(b) Yes. Let f' be a flow in the new network shown for the exercise. With the c'_i edges removed and the P'_i pumps converted back to depots. If we eliminate these edges from f' we obtain a flow f in the network of Figure 6.6. We'll have value(f) = value(f') because the sum of the net flows out of D_1 and D_2 for f equals the net flow out of D_0 for f' because $b(P'_1) = b(P'_2) = 0$ for f'.

6.4.7. Let f and g be two maximum flows and let $A = \mathcal{A}(f)$. By the proof of the Augmentable Path Theorem, we see that value(g) = value(f) if and only if $g(e) = c(e)$ for all $e \in \text{FROM}(A, B)$ and $g(e) = 0$ for all $e \in \text{FROM}(B, A)$. It is tempting to conclude that therefore $A = \mathcal{A}(g)$, but this does not follow immediately.

Here is a correct proof. As above, let $A = \mathcal{A}(f)$. If $A \neq \mathcal{A}(g)$, we can assume that there is some $v \in \mathcal{A}(g)$ with $v \notin A$. (If not, interchange the names of f and g.) Let $u_1, u_2, \ldots$ be an augmentable path for g that ends at v. Let δ be its increment. Since $v \notin A$, and $u_1 \in \mathcal{D}_{\text{in}} \subseteq A$, there is an i with $u_i \in A$ and $u_{i+1} \in B$. If $e = (u_i, u_{i+1})$ is the directed edge of G, then $g(e) \leq c(e) - \delta$ and $e \in \text{FROM}(A, B)$. If $e = (u_{i+1}, u_i)$ is the directed edge of G, then $g(e) \geq \delta$ and $e \in \text{FROM}(B, A)$. In either case, the idea in the previous paragraph proves that value(g) < value(f), contradicting the assumption that g is a maximum flow.

6.4.9 (a) This is trivial.

(b) Consider the sets when a is removed from them and the set A_n is removed. We have reduced n by 1 and (6.12) still holds (but may the inequalities may not be strict). By induction, we are done.

(c) By induction, there is an SDR for the A_i, $i \in I$. If the claimed inequality is true, then there is also an SDR for the B_i, $i \in \underline{n} - X$. Taken together, these give us our representatives. It remains to prove the inequality. We have

$$\bigcup_{i \in I \cup R} A_i \;=\; \left(\bigcup_{i \in R} B_i \right) \cup X,$$

where the last union is disjoint. Thus

$$\left| \bigcup_{i \in R} B_i \right| \;=\; \left| \bigcup_{i \in R \cup I} A_i \right| - |X| \;\geq\; |R \cup I| - |I| \;=\; |R|.$$

6.4.11. The result in the previous exercise is valid when all edges are taken to be undirected. To see this, construct a directed graph by replacing each edge $\{x, y\}$ of G with the two edges (x, y) and (y, x). The first part of the previous proof goes through. If a directed path $e_1, e_2, \ldots$ is constructed from a flow, replace each edge (x, y) in the directed path with $\{x, y\}$. This gives what we will call a pseudo-path. The same edge may appear twice in the pseudo-path because there may be two directed edges $e_i = (x, y)$ and $e_j = (y, x)$ which give the same undirected edge. We may assume that $i < j$. Replace the pseudo-path with the pseudo-path obtained from $e_1, \ldots, e_{i-1}, e_{j+1}, \ldots$. Iterating this process eventually leads to a path from u to v. (You may want to fill in some details about that.)

Section 6.5

6.5.1 (a) The probability that a vertex v has degree d is $\binom{n-1}{d}p^d(1-p)^{n-1-d}$ since we must choose d of the remaining $n-1$ vertices to connect to v, then multiply by the probability of an edge being present (p) or absent ($1-p$). Probabilities multiply since edges are independent in $\mathcal{G}_p(n)$. Using linearity of expectation and summing over all n vertices, we get $n\binom{n-1}{d}p^d(1-p)^{n-1-d}$.

(b) If C is a potential 4-cycle of 4 vertices, let $X_C = 1$ if the cycle is present and $X_C = 0$ if it is not. Then $\mathbf{E}(X_C) = p^4$. We must multiply this by the number of choices for C; that is, the number of potential 4-cycles. This number is $\binom{n}{4} \times 3 = \frac{n(n-1)(n-2)(n-3)}{4\times 2}$, which can be derived in at least two ways:

- Note that there are 3 ways to make a 4-cycle out of a set of 4 vertices.
- Choose an ordered list of 4 vertices that represent walking around a cycle. There are 4 vertices that could have been chosen as the starting vertex and 2 ways we could have gone around the cycle.

(c) This is the same as the previous situation, except that now we must make sure the two edges that cut across the 4-cycle are not present. Hence the answer is $3\binom{n}{4}p^4(1-p)^2$.

6.5.3 (a) The probability of a cycle is the probability of the union of the sets $\mathcal{G}_C$. The probability of the union of sets, is less than or equal to the sum of their separate probabilities; that is, $\Pr(A \cup B \cup \cdots) \leq \Pr(A) + \Pr(B) + \cdots$.

(b) The denominator is $|\mathcal{G}(n,k)|$. The numerator counts graphs as follows. There are $(c-1)!$ directed cycles. Since each cycle can be made directed in two ways, there are $(c-1)!/2$ cycles. Since we have used up c edges making the cycle, we must choose $k-c$ edges from the remaining $N-c$ unused edges.

(c) Collect terms in (a) according to $c = |C|$ and use (b). There are $\binom{n}{c}$ c-subsets of $\underline{n}$.

(d) The left side comes from writing $\binom{x}{m} = \frac{x(x-1)\cdots(x-m+1)}{m!}$ and doing some algebra. The inequality comes from $\frac{k!}{(k-c)!} < k^c$ and $\frac{x-j}{y-j} < \frac{x}{y}$ when $y > x \geq j$.

6.5.5 (a) Let T contain a close to half the vertices as possible. If $|V| = 2n$, $|T| = n$ and $|V-T| = n$. Since G contains all edges, this choice of T gives us a bipartite subgraph with n^2 edges. When $|V| = 2n+1$, we take $|T| = n$ and $|V-T| = n+1$, obtaining a bipartite subgraph with $n(n+1)$ edges.

(b) The example bound is $|E|/2$ and $|E| = |\mathcal{P}_2(V)| = |V|(|V|-1)/2$. For $|V| = 2n$, we have $|E|/2 = n(2n-1)/2 = n^2 - n/2$. Hence the bound is off by $n/2$. This may sound large, but the relative error is small: Since $(n^2 - n/2)/n^2 = 1 - 1/2n$, the relative error is $1/|V|$. We omit similar calculations for $|V| = 2n+1$.

(c) The idea is to construct the largest possible complete graph and then add edges in any manner whatsoever. Let m be the largest integer such that $k \geq \binom{m}{2}$, choose $S \subseteq V$ with $|S| = m$, construct a complete graph on m vertices using $\binom{m}{2}$ edges, and insert the remaining $k - \binom{m}{2}$ edges in any manner to form a simple graph $G(V,E)$. By (a), the number of edges in a bipartite subgraph of the complete graph on T has at least $(m/2)^2 - m$ edges for some constant C Since m is as large as possible, $k < \binom{m+1}{2} < \frac{(m+1)^2}{2}$. Thus $m+1 > \sqrt{2k}$. Also, since $k \geq \binom{m}{2} > \frac{(m-1)^2}{2}$, $m-1 < \sqrt{2k}$. Hence the number of edges in bipartite subgraph is at least

$$(m/2)^2 - m > \frac{\left(\sqrt{2k} - 1\right)^2}{4} - \sqrt{2k} - 1,$$

Which equals k minus terms involving $k^{1/2}$ and constants.

	σ	•	E	δ	comments on state
b	s1	sd	z	1a	starting
s1	z	sd	z	1a	part 1 sign seen
1a	z	2	e	1a	part 1 digits seen; accepting
sd	z	z	z	1b	decimal seen, no digits yet
1b	z	z	e	1b	part 1 after decimal; accepting
e	s2	z	z	2	E seen
s2	z	z	z	2	part 2 sign seen
2	z	z	z	2	part 2 digits seen; accepting
z	z	z	z	z	error seen

Figure S.6.1 The transition table for a finite automaton that recognizes floating point numbers, the possible inputs are sign (σ), decimal point (•), digit (δ) and exponent symbol (E). The comments explain the states.

(d) Call the colors 1,2,3. Let V_i be the set of vertices colored with color i and let $E_{i,j}$ be the set of edges in G that connect vertices in V_i to vertices in V_j. Since $|E| = |E_{0,1}| + |E_{0,2}| + |E_{1,2}|$, at least one of $|E_{i,j}|$ is at most $|E|/3$. Suppose it is $E_{1,2}$. The bipartite subgraph whose edges connect vertices in V_0 to vertices in $V_1 \cup V_2$ contains $E - |E_{1,2}| \geq 2|E|/3$ edges.

Section 6.6

6.6.1. The left column gives the input and the top row the states.

	0	1	2	3	4
0	0	2	4	1	3
1	1	3	0	2	4

6.6.3. The states are 0, O1, E1 and R. In state 0, a zero has just been seen; in O1, an odd number of ones; in E1, an even number. The start state is 0 and the accepting states are 0 and O1. The state R is entered when we are in E1 and see a 0. Thereafter, R always steps to R regardless of input. You should be able to finish the machine.

6.6.5. In our input, we let δ stand for any digit, since the transition is independent of which digit it is. Similarly, σ stands for any sign. There is a bit of ambiguity as to whether the integer after the E must have a sign. We assume not. The automaton contains three states that can transit to themselves: recognizing digits before a decimal, recognizing digits after a decimal and recognizing digits after the E. We call them 1a, 1b and 2. There is a bit of complication because of the need to assure digits in the first part and, if it is present, in the second part. The transition table is given in Figure S.6.1.

6.6.7 (a) We need states that keep track of how much money is held by the machine. This leads us to states named $0, 5, \ldots, 30$. The output of the machine will be indicated by An, Bn, Cn and n, where n indicates the amount of money returned and A, B and C indicate the item delivered. There may be no output. The start state is 0.

(b) See Figure S.6.2.

	5	10	25	A	B	C	R
0	5	10	25	0	0	0	0
5	10	15	30	5	5	5	0, R5
10	15	20	10, R25	10	10	10	0, R10
15	20	25	15, R25	0, A0	15	15	0, R15
20	25	30	20, R25	0, A5	0, B0	20	0, R20
25	30	25, R10	25, R25	0, A10	0, B5	0, C0	0, R25
30	30, R5	30, R10	30, R25	0, A15	0, B10	0, C5	0, R30

Figure S.6.2 The transitions and outputs for an automaton that behaves like a vending machine. The state is the amount of money held and the input is either money, a purchase choice (A, B, C) or a refund request (R).

Section 7.1

7.1.1. $\mathcal{A}(m)$ (note m, not n) is the statement of the rank formula. The inductive step and use of the inductive hypothesis are clearly indicated in the proof.

7.1.3. Let $\mathcal{A}(k)$ be the assertion that the coefficient of $y_1^{m_1}\cdots y_k^{m_k}$ in $(y_1+\cdots+y_k)^n$ is $n!/m_1!\cdots m_k!$ if $n = m_1+\cdots+m_k$ and 0 otherwise. $\mathcal{A}(1)$ is trivial. We follow the hint for the induction step. Let $x = y_1+\cdots+y_{k-1}$. By the binomial theorem, the coefficient of $x^m y_k^{m_k}$ in $(x+y_k)^n$ is $n!/m!m_k!$ if $n = m+m_k$ and 0 otherwise. By the induction hypothesis, the coefficient of $y_1^{m_1}\cdots y_{k-1}^{m_{k-1}}$ in x^m is $m!/m_1!\cdots m_{k-1}!$ if $m = m_1+\cdots+m_{k-1}$ and zero otherwise. Combining these results we see that the coefficient of $y_1^{m_1}\cdots y_k^{m_k}$ in $(y_1+\cdots+y_k)^n$ is

$$\frac{n!}{m!\,m_k!}\,\frac{m!}{m_1!\cdots m_{k-1}!}$$

if $n = m_1+\cdots+m_k$ and 0 otherwise.

7.1.5 (a) $x_1'x_2' + x_1'x_2 \;=\; x_1'$.

(b) $x_1'x_2 + x_1x_2'$.

(c) $x_1'x_2'x_3 + x_1'x_2x_3 + x_1x_2'x_3' + x_1x_2x_3' \;=\; x_1'x_3 + x_1x_3'$.

(d) $x_1'x_2'x_3 + x_1'x_2x_3' + x_1'x_2x_3 + x_1x_2'x_3' \;=\; x_1'x_2 + x_1'x_3 + x_1x_2'x_3'$.

7.1.7. If you are familiar with de Morgan's laws for complementation, you can ignore the hint and give a simple proof as follows. By Example 7.3, one can express f' in disjunctive form: $f' = M_1 + M_2 + \cdots$. Now $f = (f')' = M_1'M_2'\cdots$ by de Morgan's law and, if $M_i = y_1y_2\cdots$, then $M_i' = y_1'+y_2'+\cdots$ by de Morgan's law.

To follow the hint, replace (7.5) with

$$f(x_1,\ldots,x_n) \;=\; (g_1(x_1,\ldots,x_{n-1}) + x_n')\,(g_0(x_1,\ldots,x_{n-1}) + x_n)$$

and practically copy the proof in Example 7.3.

7.1.9. We can induct on either k or n. It doesn't matter which we choose since the formula we have to prove is symmetric in n and k. We'll induct on n. The given formula is $\mathcal{A}(n)$. For $n = 0$, the

formula becomes $F_{k+1} = F_{k+1}$, which is true.

$$\begin{aligned} F_{n+k+1} &= F_{(n-1)+(k+1)+1} && \text{using the hint} \\ &= F_nF_{k+2} + F_{n-1}F_{k+1} && \text{by } \mathcal{A}(n-1) \\ &= F_n(F_{k+1} + F_k) + F_{n-1}F_{k+1} && \text{by definition of } F_{k+2} \\ &= (F_n + F_{n-1})F_{k+1} + F_nF_k && \text{by rearranging} \\ &= F_{n+1}F_{k+1} + F_nF_k && \text{by definition of } F_{n+1}. \end{aligned}$$

Section 7.2

7.2.1. Given that p and q are positive integers, it does not follow that p' and q' are positive integers. (For example, let $p = 1$.) Thus $\mathcal{A}(n-1)$ may not apply.

7.2.3. You may object that the induction has not been clearly phrased, but this can be overcome: Let I be the set of interesting positive integers and let $\mathcal{A}(n)$ be the assertion $n \in I$. If $\mathcal{A}(1)$ is false, then even 1 is not interesting, which is interesting. The inductive step is as given in the problem: If $\mathcal{A}(n)$ is false, then since $\mathcal{A}(k)$ is true for all $k < n$, n is the smallest uninteresting number, which is interesting.

Then what *is* wrong? It is unclear what "interesting" means, so the set of interesting positive integers is not a well defined concept. Proofs based on foggy concepts are always suspect.

7.2.5. To show the equivalence, we must show that an object is included in one definition if and only if it is included in the other. We do this by induction on the number of vertices. Before doing this, however, we observe that the objects constructed in Example 7.9 are trees:

- They are connected since $T_1, \ldots, T_k$ are connected by induction.
- They have no cycles since $T_1, \ldots, T_k$ have no cycles by induction and have no vertices in common by assumption.

We now turn to the inductive proof of equivalence.

(i) You should be able to see that both definitions include the single vertex.

(ii) Now for the inductive step in one direction: Suppose T has $n > 1$ vertices and is included in the definition in Example 7.9. By the induction hypothesis, $T_1, \ldots, T_k$ are included in Definition 5.12 (p. 139). By the construction in Example 7.9, the roots of $T_1, \ldots, T_k$ are ordered and are the children of the root of the new tree. Furthermore, joining $T_1, \ldots, T_k$ to a new root preserves the orderings and parent-child relationships in the T_i. Hence this tree satisfies Definition 5.12.

(iii) Now for the other direction. Let $r_1, \ldots, r_k$ be the ordered children of the root r of the tree T in Definition 5.12. Following on down through the children of r_i, we obtain an RP-tree T_i which is included in Example 7.9 since it has fewer than n vertices. By the argument in (ii), the construction forms an RP-tree from $T_1, \ldots, T_k$ which can be seen to be the same as T.

Section 7.3

7.3.1 (a) We must compare as long as both lists have items left in them. After all items have been removed from one list, what remains can simply be appended to what has been sorted. All items will be removed from one list the quickest if each comparison results in removing an item from the shorter list. Thus we need at least $\min(k_1, k_2)$ comparisons.

On the other hand, suppose we have $k_1 + k_2$ items and the smallest ones are in the shorter list. In this case, all the items are removed from the shorter list and none from the longer in the first $\min(k_1, k_2)$ comparisons, so we have achieved the minimum.

(b) Here's the code. Note that the two lists have lengths m and $n-m$ and that $\min(m, n-m) = m$ because $m \le n/2$.

```
Procedure c(n)
    c = 0
    If (n = 1), then Return c
    Let m be n/2 with remainder discarded
    c = c + c(m)
    c = c + c(n − m)
    c = c + m
    Return c
End
```

(c) We have $c(2^0) = 0$ and $c(2^{k+1}) = 2c(2^k) + 2^k$ for $k \ge 0$. The first few values are

$$c(2^0) = 0, \quad c(2^1) = 2^0, \quad c(2^2) = 2 \times 2^1, \quad c(2^3) = 3 \times 2^2, \quad c(2^4) = 4 \times 2^3.$$

This may be enough to suggest the pattern $c(2^k) = k \times 2^{k-1}$; if not, you can compute more values until the pattern becomes clear.

We prove it by induction. The conjecture $c(2^k) = k \times 2^{k-1}$ is the induction assumption. For $k = 0$, we have $c(2^0) = 0$, and this is what the formula gives. For $k > 0$, we use the recursion to reduce k and then use the induction assumption:

$$c(2^k) \;=\; 2c(2^{k-1}) + 2^{k-1} \;=\; 2 \times (k-1) \times 2^{k-2} + 2^{k-1} \;=\; k \times 2^{k-1},$$

which completes the proof.

When k is large,

$$\frac{c(2^k)}{C(2^k)} \;=\; \frac{k \times 2^{k-1}}{(k-1)2^k + 1} \;=\; \frac{k/2}{k - 1 + 2^{-k}} \;\sim\; 1/2.$$

This shows that the best case and worst case differ by about a factor of 2, which is not very large.

7.3.3. Here is code for computing the number of moves.

```
Procedure M(n)
    M = 0
    If (n = 1), then Return M
    Let m be n/2 with remainder discarded
    M = M + M(m)
    M = M + M(n − m)
    M = M + n
    Return M
End
```

This gives us the recursion $M(2^k) = 2M(2^{k-1}) + 2^k$ for $k > 0$ and $M(2^0) = 0$. The first few values are

$$M(2^0) = 0, \quad M(2^1) = 2^1, \quad M(2^2) = 2 \times 2^2, \quad M(2^3) = 3 \times 2^3, \quad M(2^4) = 4 \times 2^4.$$

Thus we guess $M(2^k) = k2^k$, which can be proved by induction.

7.3.5 (a) Here's one possible procedure. Note that the remainder must be printed out *after* the recursive call to get the digits in the proper order. Also note that one must be careful about zero: A string of zeroes should be avoided, but a number which is zero should be printed.

```
OUT(m)
     If m < 0, then
          Print ''−''
          Set m = −m
     End if
     Let q and 0 ≤ r ≤ 9 be determined by m = 10q + r
     If q > 0, then OUT(q)
     Print r
End
```

(b) Single digits

(c) When `OUT` calls itself, it passes an argument that is smaller in magnitude than the one it received, thus `OUT(`m`)` must terminate after at most $|m|$ calls.

7.3.7. The description for $k = 1$ is on the left and that for $k > 1$ is on the right:

$$\underline{n^1} \qquad\qquad\qquad\qquad \underline{n^k}$$

$$1 \quad \cdots \quad n \qquad\qquad k,\ \underline{k-1^{k-1}} \quad \cdots \quad n,\ \underline{n-1^{k-1}}$$

7.3.9 (a) Let $\mathcal{A}(n)$ be the assertion "$\mathtt{H}(n, S, E, G)$ takes the least number of moves." Clearly $\mathcal{A}(1)$ is true since only one move is required. We now prove $\mathcal{A}(n)$. Note that to do $S \xrightarrow{n} G$ we must first move all the other washers to pole E. They can be stacked only one way on pole E, so moving the washers from S to E requires using a solution to the Tower of Hanoi problem for $n-1$ washers. By $\mathcal{A}(n-1)$, this is done in the least number of moves by $\mathtt{H}(n-1, S, G, E)$. Similarly, $\mathtt{H}(n-1, E, S, G)$ moves these washers to G in the least number of moves.

(b) Simply replace $\mathtt{H}(m, \ldots)$ with $S(m)$ and replace a move with a 1 and adjust the code a bit to get

```
Procedure S(n)
     If (n = 1)  Return 1.
     M = 0
     M = M + S(n − 1)
     M = M + 1
     M = M + S(n − 1)
     Return M
End
```

The recursion is $S(1) = 1$ and $S(n) = 2S(n-1) + 1$ when $n > 1$.

(c) The values are 1, 3, 7, 15, 31, 63, 127.

(d) Let $\mathcal{A}(n)$ be "$S(n) = 2^n - 1$." $\mathcal{A}(1)$ asserts that $S(1) = 1$, which is true. By the recursion and then the induction hypothesis we have

$$S(n) \;=\; 2S(n-1)+1 \;=\; 2(2^{n-1}-1)+1 \;=\; 2^n - 1.$$

(e) By studying the binary form of k and the washer moved for small n (such as $n = 4$) you could discover the following rule.

> If $k = \cdots b_3 b_2 b_1$ is the binary representation of k, $b_j = 1$,
> and $b_i = 0$ for all $i < j$, then washer j is moved.

(This simply says that b_j is the lowest nonzero binary digit.) No proof was requested, but here's one. Let $\mathcal{A}(n)$ be the claim for $\mathtt{H}(n,\ldots)$. $\mathcal{A}(1)$ is trivial. We now prove $\mathcal{A}(n)$. If $k < 2^{n-1}$, it follows from $S(m)$ that $\mathtt{H}(n-1,\ldots)$ is being called and $\mathcal{A}(n-1)$ applies. If $k = 2^{n-1}$, then we are executing $S \xrightarrow{n} G$ and so this case is verified. Finally, if $2^{n-1} < k < 2^n$, then $\mathtt{H}(n-1,\ldots)$ is being executed at step $k - 2^{n-1}$, which differs from k only in the loss of its leftmost binary bit.

(f) Suppose that we are looking at move $k = \cdots b_3 b_2 b_1$ and that washer j is being moved. (That means b_j is the rightmost nonzero bit.) You should be able to see that this is move number $\cdots b_{j+2} b_{j+1} = (k - 2^{j-1})/2^j$ for the washer. Call this number k'. To determine source and destination, we must study move patterns.

The pattern of moves for a washer is either

$$P_0\colon S \to G \to E \to S \to G \to E \to \cdots \text{ repeating or}$$
$$P_1\colon S \to E \to G \to S \to E \to G \to \cdots \text{ repeating.}$$

Which washer uses which pattern? Consider washer j it is easily verified that it is moved a total of 2^{n-j} times, after which time it must be at G. A washer following P_i is at G only after move numbers of the form $3t + i + 1$ for some t. Thus $i + i$ is the remainder when 2^{n-j} is divided by 3. The remainder is 1 if $n - j$ is even and 0 otherwise. Thus washer j follows pattern P_i where i and $n - j$ have the same parity. If we look at the remainder after dividing k' by 3, we can see what the source and destination are by looking at the start of P_i. For those of you familiar with congruences, the remainder is congruent to $(-1)^j k + 1$ modulo 3.

7.3.11 (a) We have

$$H^*(n, S, E, G)$$

$$H^*(n-1, S, E, G) \quad S \xrightarrow{n} E \quad H^*(n-1, G, E, S) \quad E \xrightarrow{n} G \quad H^*(n-1, S, E, G)$$

(b) The initial condition is $h_1^* = 2$. For $n > 1$ we have $h_n^* \;=\; 3h_{n-1}^* + 2$.
Alternatively, $h_0^* = 0$ and, for $n > 0$, $h_n^* \;=\; 3h_{n-1}^* + 2$.

(c) The general solution is $h_n^* \;=\; 3^n - 1$. To prove it, use induction. First, it is correct for $n = 0$. Then, for $n > 0$,

$$h_n^* \;=\; 3h_{n-1}^* + 2 \;=\; 3(3^{n-1}-1) + 2 \;=\; 3^n - 1.$$

7.3.13 (a) We omit the picture.

(b) Induct on n. It is true for $n = 1$. If $n > 1$, $a_2, \ldots, a_n \in G(k_2, \ldots, k_n)$ by the induction hypothesis. Thus $a_1, a_2, \ldots, a_n$ is in a_1, H and $a_1, R(H)$.

(c) Induct on n. It is true for $n = 1$. Suppose $n > 1$ and let the adjacent leaves be $b_1, \ldots, b_n$ and $c_1, \ldots, c_n$, with c following b. If $b_1 = c_1$, then apply the induction hypothesis to $G(k_2, \ldots, k_n)$ and the sequences $b_2, \ldots, b_n$ and $c_2, \ldots, c_n$. If $b_1 \neq c_1$, it follows from the local description that $c_1 = b_1 + 1$, that $b_2, \ldots, b_n$ is the rightmost leaf in H (or $R(H)$) and that $c_2, \ldots, c_n$ is the leftmost leaf in $R(H)$ (or H, respectively). In either case, $b_2, \ldots, b_n$ and $c_2, \ldots, c_n$ are equal because they are the same leaf of H.

(d) Let $R_n(\alpha)$ be the rank of $\alpha_1, \ldots, \alpha_n$. Clearly $R_1(\alpha) = \alpha_1 - 1$. If $n > 1$ and $\alpha_1 = 1$, then $R_n(\alpha) = R_{n-1}(\alpha_2, \ldots, \alpha_n)$. If $n > 1$ and $\alpha_1 = 2$, then $R_n(\alpha) = 2^n - 1 - R_{n-1}(\alpha_2, \ldots, \alpha_n)$. Letting $x_i = \alpha_i - 1$, we have $R_n(\alpha) = (2^n - 1)x_1 + (-1)^{x_1} R_{n-1}(\alpha_2, \ldots, \alpha_n)$ and so

$$R_n(\alpha) = (2^n-1)x_1+(-1)^{x_1}(2^{n-1}-1)x_2+(-1)^{x_1+x_2}(2^{n-2}-1)x_3+\cdots+(-1)^{x_1+x_2+\cdots+x_{n-1}}x_n.$$

(e) If you got this, congratulations. Let j be as large as possible so that $\alpha_1, \ldots, \alpha_j$ contains an even number of 2's. Change α_j. (Note: If $j = 0$, $\alpha = 2, 1, \ldots, 1$, the sequence of highest rank, and it has no successor.)

Section 7.4

7.4.1. Let $M(n)$ be the minimum number of multiplications needed to compute x^n. We leave it to you to verify the following table for $n \leq 9$

n	2	3	4	5	6	7	8	9	15	21	47	49
$M(n)$	1	2	2	3	3	4	3	4	5	6	8	7

Since $15 = 3 \times 5$, it follows that $M(15) \leq M(3) + M(5) = 5$. Likewise, $M(21) \leq M(3) + M(7) = 6$. Since the binary form of 49 is 110001_2, $M(49) \leq 7$. Since $47 = 101111_2$, we have $M(47) \leq 9$, but we can do better. Using $47 = 2 \times 23 + 1$, gives $M(47) \leq M(23) + 2$, which we leave for you to work out. A better approach is given by $47 = 5 \times 9 + 2$. Since x^2 is computed on the way to finding x^5, it is already available and so $M(49) \leq M(5) + M(9) + 1 = 8$. It turns out that these are minimal, but we will not prove that.

7.4.3. Let $\vec{v}_n$ be the transpose of $(a_n, \ldots, a_{n+k-1})$. Then $\vec{v}_n = M\vec{v}_{n-1}$ where

$$M = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ a_k & a_{k-1} & a_{k-2} & \cdots & a_1 \end{pmatrix}.$$

7.4.5. Finding a maximum of n items can be done in $\Theta(n)$, so it's the computation of all the different $F(v)$ values is the problem. Thus we could compute the values of F separately from finding the maximum. However, since it's convenient to compute the maximum while we're computing the values of F, we'll do it.

The root r of T has two sons, say s_L and s_R. Observe that the answer for the tree rooted at r must be either the answer for the tree rooted at s_L or the answer for the tree rooted at s_R or $F(r)$. Also

$$F(r) = f(r) + F(s_L) + F(s_R).$$

Here's an algorithm that carries out this idea.

```
/* r is the root of the tree in what follows. */


Procedure BestSum(r)
     Call Recur(r, Fvalue, best)
     Return best
End


Procedure Recur(r, F, best)
     If r is a leaf then
          F = f(r)
          best = f(r)
     Else
          Let s_L and s_R be the sons of r.
          Recur(s_L, F_L, b_L)
          Recur(s_R, F_R, b_R)
          F = f(r) + F_L + F_R
          best = max(F, b_L, b_R)
     End if
     Return
End
```

Since $f(r)$ is only used once, the running time of this algorithm is $\Theta(n)$ for n vertices, an improvement over $\Theta(n \ln n)$.

7.4.7 (a) We can use induction: It is easily verified for $n = 1$ and $n = 2$. For $n > 2$ we have

$$\begin{aligned} a_n &= a_{n-1} + a_{n-2} = (a_0 F_{n-2} + a_1 F_{n-3}) + (a_0 F_{n-3} + a_1 F_{n-4}) \\ &= a_0(F_{n-2} + F_{n-3}) + a_1(F_{n-3} + F_{n-4}) = a_0 F_{n-1} + a_1 F_{n-2}. \end{aligned}$$

(b) Since the a_i satisfy the same recursion as the Fibonacci numbers, it is easily seen that the a sequence is just the F sequence shifted by k.

Section 8.1

8.1.1. This is exactly the situation in the text, *except* that there is now one additional question when one reaches a leaf.

8.1.3 (a) If T is not a full binary tree, there is some vertex v that has only one child, say s. Shrink the edge (v, s) so that v and s become one vertex and call the new tree T'. If there are k leaves in the subtree whose root is v, then $\mathrm{TC}(T') = \mathrm{TC}(T) - k$.

(b) We follow the hint. Let k be the number of leaves in the subtree rooted at v. Since T is a binary tree and v is not a root, $k \geq 2$ Let $d = h(v) - h(l_2)$ and note that $d = (h(l_1) - 1) - h(l_2) \geq 1$. The distance to the root of every vertex in the subtree rooted at v is decreased by d and the distance of l_2 to the root is increased by d. Thus TC is decreased by $kd - d > 0$.

(c) By the discussion in the proof of the theorem, we know that the height of T must be at least m because a binary tree of height $m-1$ or less has at most 2^{m-1}. Suppose T had height $M > m$. By the previous part of this exercise, the leaves of T have heights M and, perhaps, $M - 1$. Thus, every vertex of v with $h(v) < M-1$ has two children. It follows that T has 2^{M-1} vertices w with $h(w) = M - 1$. If these were all leaves, T would have $2^{M-1} \geq 2^m$ leaves; however, at least one vertex u with $d(u) = M - 1$ is not a leaf. Since it has two children, T has at least $2^m + 1$ leaves, a contradiction.

(d) By the previous two parts, all leaves of T have height at most m. If T' is principal subtree of T, its leaves have height at most $m - 1$ in T'. Hence T' has at most 2^{m-1} leaves.

The argument hints at how to construct the desired tree: Construct T', a principal subtree of T, having all its leaves at height $m - 1$ in T'. Construct a binary tree T'' having $n - 2^{m-1}$ such that $\mathrm{TC}(T'')$ is as small as possible. The principal subtrees of T will be T' and T''.

8.1.5 (a) Suppose the answer is S_n. Clearly $S_0 = 1$ since the root is the only vertex. We need a recursion for S_n. One approach is to look at the two principal subtrees. Another is to look at what happens when we add a new "layer" by replacing each leaf with

For the first approach, $S_n = 1 + 2S_{n-1}$, where each S_{n-1} is due to a principal subtree and the 1 is due to the root. The result follows by induction:

$$S_n \;=\; 1 + 2S_{n-1} \;=\; 1 + 2(2^n - 1) \;=\; 2^{n+1} - 1.$$

For the second approach, $S_n = S_{n-1} + 2^n$ and so $S_n = (2^n - 1) + 2^n = 2^{n+1} - 1$. By the way, if we have both recursions, we can avoid induction since we can solve the two equations

$$S_n \;=\; 1 + 2S_{n-1} \qquad \text{and} \qquad S_n \;=\; S_{n-1} + 2^n$$

to obtain the formula for S_n. Thus, by counting in two ways (the two recursions), we don't need to be given the formula ahead of time since we can solve for it.

(b) Let the value be $\mathrm{TC}^*(n)$. Again, we use induction and there are two approaches to obtaining a recursion. Clearly $\mathrm{TC}^*(1) = 0$, which agrees with the formula.

The first approach to a recursion: Since the principal subtrees of T each store S_{n-1} keys and since the path lengths all increase by 1 when we adjoin the principal subtrees to a new root, $\mathrm{TC}^*(n) \;=\; 2(S_{n-1} + \mathrm{TC}^*(n-1))$. Thus

$$\mathrm{TC}^*(n) \;=\; 2(2^n - 1 + (n-2)2^n + 2) \;=\; 2((n-1)2^n + 1) \;=\; (n-1)2^{n+1} + 2.$$

For the second approach, $\mathrm{TC}^*(n) = \mathrm{TC}^*(n-1) + n2^n$. Again, we can prove the formula for $\mathrm{TC}^*(n)$ by induction or, as in (a), we can solve the two recursions directly.

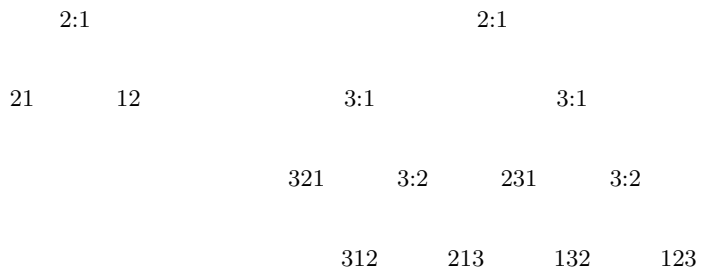


Figure S.8.1 The decision trees for binary insertion sorts. Go to the left at vertex $i:j$ if $u_i < s_j$ and to the right otherwise. (You may have done the reverse and gotten the mirror images. That's fine.)

Section 8.2

8.2.1. Here are the first few and the last.

1. Start the sorted list with 9.
2. Compare 15 with 9 and decide to place it to the right giving 9, 15.
3. Compare 6 with 9 to get 6, 9, 15.
4. Compare 12 with 9 and then with 15 to get 6, 9, 12, 15.
5. Compare 3 with 9 and then with 6 to get 3, 6, 9, 12, 15.

. . . . . .

16. We now have the sorted list 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 16. Compare 8 with 7, with 12 with 10 and then with 9 to decide where it belongs.

8.2.3. See Figure S.8.1. To illustrate, suppose the original list is 3,1,2. Thus $u_1 = 3$, $u_2 = 1$ and $u_3 = 2$.

- We start by putting u_1 in the sorted list, so we have $s_1 = 3$.
- Now u_2 must be inserted into the list s_1. We compare u_2 with s_1, the 2:1 entry. Since $s_1 = 3 > 1 = u_2$, we go to the left and our sorted list is $1, 3$ so now $s_1 = 1$ and $s_2 = 3$.
- Now u_3 must be inserted into the list s_1, s_2. Since we are at 3:1, we compare $u_3 = 2$ with $s_1 = 1$ and go to the right. At this point we know that u_3 must be inserted into the list s_2. We compare $u_3 = 2$ with $s_2 = 3$ at 3:2 and go to the left.

8.2.5 (a) Suppose that the alphabet has L letters and let the ith letter (in order) be a_i. Let u_j be a word with exactly k letters. The following algorithm sorts $u_1, \ldots, u_n$ and returns the result as $x_1, \ldots, x_n$.

```
BUCKET(u_1, ..., u_n)
     Copy u_1, ..., u_n to x_1, ..., x_n.
     /* t is the position in the word. */
     For t = k to 1
     /* Make the buckets. */
          Create L empty ordered lists.
          For j = 1 to n
               If the tth letter of x_j is a_i,
                    then place x_j at the end of the ith list.
               End for
```

```
            Copy the ordered lists to x_1, ..., x_n, starting with
                 the first item in the first list and ending with the
                 last item in the Lth list.
            End for
End
```

(b) Extend all words to k letters by introducing a new letter called "blank" which precedes all other letters alphabetically. Apply the algorithm in (a).

8.2.7. First divide the list into two equally long tapes, say

A: 9, 15, 6, 12, 3, 7, 11 5 B: 14, 1, 10, 4, 2, 13, 16, 8.

Think of each tape as containing a series of 1 long (sorted) lists. (The commas don't appear on the tapes, they're just there to help you see where the lists end.) Merge the first half of each tape, list by list, to tape C and the last halves to D. This gives us the following tapes containing a series of 2 long sorted lists:

C: 9 14, 1 15, 6 10, 4 12 D: 2 3, 7 13, 11 16, 5 8.

Now we merge these 2 long lists to get 4 long lists, writing the results on A and B:

A: 2 3 9 14, 1 7 13 15 B: 6 10 11 16, 4 5 8 12.

Merging back to C and D gives

C: 2 3 6 9 10 11 14 16 D: 1 4 5 7 8 12 13 15.

These are merged to produce one 16 long list on A and nothing on B.

8.2.9. A split requires $n-1$ comparisons since the chosen item must be compared with every other item in the list In the worst case, we may split an n long list into one of length 1 and another of length $n-1$. We then apply Quicksort to the list of length $n-1$. If $W(n)$ comparisons are needed, then $W(1)=0$ and $W(n) = n-1+w(n-1)$ for $n>1$. Thus $W(n) = \sum_{k=1}^{n-1} k = n(n-1)/2$.

Suppose that $n=2^k$ and the lists are split evenly. Let $E(k)$ be the number of comparisons. Since Quicksort is applied to two lists of length $n/2$ after splitting, $E(k) = n-1+2E(k-1)$ for $k>0$ and $E(0)=0$. A little computation gives us $E(1)=1$, $E(2)=5$, $E(3)=17$, $E(4)=49$ and $E(5)=129$. From the statement of the problem we expect $E(k)$ to be near $k2^k$, which has the values 0, 2, 8, 24 and 64. Comparing these sequences we discover that $E(k) = 2(k-1)2^{k-1}+1$ for $k<6$. This is easily proved by induction

Section 8.3

8.3.1. Since there are only 3 things, you cannot compare more than one pair of things at any time. By the Theorem 8.1, we need at least $\log_2(3!)$ comparisons; i.e., at least three. A network with three comparisons that sorts is given in Figure 8.2.

8.3.3. As argued in the previous two solutions, we will need at least seven comparisons and we can do at least two per time. This means it will take at least four time units. It has been shown (but not in this text!) that at least five time units are required. A brick wall sort works.

8.3.5. One possibility is the type of network shown in Figure 8.3. For n inputs, this has

$$1 + 2 + \cdots + (n-1) \;=\; \frac{n(n-1)}{2}$$

comparators. It was noted in the text that a brick wall for n items must have length n. If n is even there are $(n/2)(n-1)$ comparators and if n is odd there are $n((n-1)/2)$ comparators. Thus this is the same as Figure 8.3. We don't know if it can be done with less.

8.3.7. By the Adjacent Comparisons Theorem, we need only check the sequence $n, \ldots, 2, 1$. Using the argument that proves the Zero-One Principle, it follows that this sequence is sorted if and only if all sequences that consist of a string of ones followed by a string of zeroes are sorted.

8.3.9. It is evident that the idea in the solution to (a) of the previous exercise works for any n. This can be used as the basis of an inductive proof.

An alternative proof can be given using sequences that consist of ones followed by zeroes. (See Exercise 8.3.7.) Note that when the lowest 1 starts moving down through the comparators, it moves down one line each time unit until it reaches the bottom. The 1 immediately above it starts the same process one time unit later. The 1 immediately above this one starts one more time unit later, and so forth. If there are j ones and the lowest 1 reaches the bottom after an exchange at time t, then the next 1 reaches proper position after an exchange at time $t+1$. Continuing in this way, all ones are in proper position after the exchanges at time $t+j-1$. Suppose the jth 1 (i.e., lowest 1) starts moving by an exchange at time i. Since it reaches position after $n-j$ exchanges, $t \;=\; i+(n-j)-1$. Thus all ones are in position after the exchanges at time $(i+(n-j)-1)+j-1 \;=\; n+i-2$. The jth 1 starts moving when it is compared with the line below it. This happens at time 1 or 2. Thus $n+i-2 \;\leq\; n$.

8.3.11. Use induction on n. For $n = 2^1$, it works. Suppose that it works for all powers of 2 less than 2^t. We use the variation of the Zero-One Principle mentioned in the text. Suppose that the first half of the x_i's contains α zeroes and the second half contains β zeroes. `BMERGE` calls `BMERGE2` with $k = j = 2^{t-1}$ By the induction assumption, `BMERGE2` rearranges the "odd" sequence $x_1, x_3, \ldots, x_{2^t-1}$ in order and the "even" sequence $x_2, x_4, \ldots, x_{2^t}$ in order. The number of zeroes in the odd sequence minus the number of zeroes in the even sequence in 0, 1 or 2; depending on how many of α and β are odd. When the difference is 0 or 1, the result of `BMERGE2` is sorted. Otherwise, the last zero in the odd sequence, $x_{\alpha+\beta+1}$, is after the first one in the even sequence, $x_{\alpha+\beta}$, and all other x_i's are in order. The comparator in `BMERGE` with $i = (\alpha+\beta)/2$ fixes this.

8.3.13 (a) Since a one long list is sorted, nothing is done and so $S(0) = 0$. The two recursive calls of `BSORT` can be implemented by a network in which they run during the same time interval. This can then be followed by the `BMERGE` and so $S(N) \;\leq\; S(N-1) + M(N)$.

(b) As for $S(0) = 0$, $M(0) = 0$ is trivial. Since all the comparators mentioned in `BMERGE` and be run in parallel at the same time, $M(N) \;\leq\; M(N-1) + 1$.

(c) From the previous part, it easily follows by induction on N that $M(N) \leq N$. Thus $S(N) \;\leq\; S(N-1) + N$ and the desired result follows by induction on N.

(d) If $2^{N-1} < n \leq 2^N$, then the above ideas show that $S(n) \leq N(N+1)/2$. Thus

$$S(n) \ < \ \frac{1}{2}\,(1+\log_2 n)(2+\log_2 n).$$

Section 9.1

9.1.1. We do PREV(T).

```
PREV(T)
     Let r be the root of T
     Let T_1, ..., T_k be the principal subtrees of T
     Output r
     For i = 1, ..., k    Prev(T_i)
End
```

9.1.3. We give pseudocode for vertex visitation.

```
BFV(T)
     Initialize queue
     INQUEUE(T)
     While queue not empty
          S = OUTQUEUE( )
          Let r be the root of S
          Let S_1, ..., S_k be the principal subtrees of S
          Output r
          For i = 1, ..., k   INQUEUE(S_i)
     End while
End
```

9.1.5. The proof can be done by induction on the size of the tree by showing that the comments in the algorithm are correct. To do this, we need to notice a couple of things.

- By removing r from G before constructing S, we guarantee that S will not contain r. Thus it will contain precisely the vertices that are reachable on a path from r, starting with the edge $\{r, s\}$.
- Because we remove the root vertex of the tree from G and do this recursively, whenever a tree is ready to return, all its vertices have been removed from G. As a result, none of the vertices in S are left in G when we construct R.

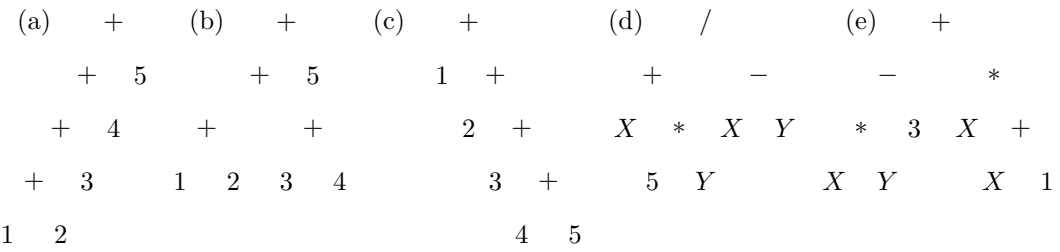
9.1.9 (a) $D(T)$ is $+1, D(T_1), -1, +1, D(T_2), -1, \ldots, +1, D(T_m), -1$.

(b) Each edge is traversed twice, proving the sum. The rest can be proved by induction on the number of vertices using the formula in (a). Actually, one can show more: The sum up to k is the length of the path from the root to the vertex that is reached after k steps.

(c) The "if" part follows from (b). The "only if" part can be done by showing that there is a unique way to construct a tree associated with such a sequence. This can be done recursively if we use the observation that the sum up to k is 0 if and only if we have returned to the root after k steps: Let k be the first index for which the sum is 0. We must have $s_1 = +1$, $s_k = -1$ and the subsequences $s_2, \ldots, s_{k-1}$ and $s_{k+1}, \ldots, s_n$ are associated with unique RP-trees. There is just one way to piece these trees together.

Section 9.2

9.2.1.



9.2.3. We use `value`$(\cdots)$ to indicate the value of a variable or constant.

(a) The first method:

```
EVALUATE(exp)
     If (exp has no op)   Return value(exp).
     If (exp = −exp1)   Return −value(exp1).
     Let exp = (exp1  op ; exp2).
     Return EVALUATE(exp1) op EVALUATE(exp2).
End
```

(b) The second method:

```
EVALUATE(T)
     Let r be the root of T.
     Let k be the number of principal subtrees of T
          and let T_i be the ith of them.
     If (k = 0), Return value(r).
     For i = 1, ..., k    Let v_i = EVALUATE(T_i).
     /* If k = 1, r should be unary minus. */
     If k = 1, Return r  v_1.
     If k = 2, Return v_1  r  v_2.
End
```

9.2.5. We will indicate what needs to be added. Other solutions are possible.

(a) *exp* → − *term*

(b) *term* → *power* and *power* → *factor* | *factor* ∗∗ *power*

(c) Let *subst* be the start symbol now and add *subst* → *exp* | **id :=** *exp*

(d) This is a bit trickier because the **:=** must reach as far to the right as possible. In particular, you cannot replace the last three items in the following list with just *factor* → *subst*. Let *start* be the start symbol.

$$
\begin{array}{rcl}
\textit{start} & \rightarrow & \textit{exp} \mid \textit{subst} \\
\textit{subst} & \rightarrow & \textbf{id :=}\ \textit{exp} \mid \textbf{id :=}\ \textit{subst} \\
\textit{exp} & \rightarrow & \textit{exp}{+}\textit{subst} \mid \textit{exp}{-}\textit{subst} \\
\textit{term} & \rightarrow & \textit{term}{*}\textit{subst} \mid \textit{term}/\textit{subst} \\
\textit{factor} & \rightarrow & (\ \textit{subst}\ )
\end{array}
$$

Section 9.3

9.3.1. The construction starts with •. The first iteration gives • and all trees that have children produced in the starting step. Thus we get

In the next iteration, we obtain the following new trees with at most 4 vertices.

In the next step, the only new tree is a 4-vertex tree consisting of a path from the root to a single leaf. After this, no new trees with less than 5 vertices are obtained.

9.3.3. For $k \le 7$, the values are in the text. $b_8 = 429$, $b_9 = 1430$ and $b_{10} = 4862$.

9.3.5. We'll use $n{:}\,r$ to mean a tree with n leaves and rank r and $(n_1{:}\,r_1,\ n_2{:}\,r_2)$ to mean a tree with left son $n_1{:}\,r_1$ and right son $n_2{:}\,r_2$. We use formula (9.5) and the greedy approach: First make $|T_1|$ as large as possible, then make $\mathrm{RANK}(T_1)$ as large as possible. Here are the calculations. You should be able to construct the trees easily from the results as long as you remember (a) that $n{:}\,0$ describes a tree in which all the left sons are leaves (since that is the leftmost tree in the list of trees) and (b) that since there is only one tree with 1 leaf and only one with 2 leaves, they each have rank 0.

$$\begin{aligned}
8{:}\,100 &= (1{:}\,0,\ 7{:}\,100) && \text{since } b_1b_7 = 132 > 100,\ \text{we have } |T_1| = 1 \text{ and } 100 = 0b_7 + 100\\
7{:}\,100 &= (6{:}\,10,\ 1{:}\,0) && \text{since } b_1b_6 + \cdots + b_5b_2 = 90 \text{ and } 10 = 10b_1 + 0\\
6{:}\,10 &= (1{:}\,0,\ 5{:}\,10) && \text{since } b_1b_5 = 14 > 10 \text{ and } 10 = 0b_5 + 10\\
5{:}\,10 &= (4{:}\,1,\ 1{:}\,0) && \text{since } b_1b_4 + \cdots + b_3b_2 = 9 \text{ and } 1 = 1b_1 + 0\\
4{:}\,1 &= (1{:}\,0,\ 3{:}\,1)\\
3{:}\,1 &= (2{:}\,0,\ 1{:}\,0)
\end{aligned}$$

$$\begin{aligned}
8{:}\,200 &= (3{:}\,1,\ 5{:}\,12) && \text{since } b_1b_7 + b_2b_6 = 174 \text{ and } 26 = 1b_5 + 12\\
5{:}\,12 &= (4{:}\,3,\ 1{:}\,0) && \text{since } b_1b_4 + b_2b_3 + b_3b_2 = 9\\
4{:}\,3 &= (3{:}\,0,\ 1{:}\,0) && \text{since } b_1b_3 + b_2b_2 = 3
\end{aligned}$$

$$\begin{aligned}
8{:}\,300 &= (7{:}\,3,\ 1{:}\,0)\\
n{:}\,3 &= (1{:}\,0,\ n-1{:}\,3) && \text{for } n \ge 5 \text{ since } b_1b_{n-1} > 3\\
4{:}\,3 &= (3{:}\,0,\ 1{:}\,0)
\end{aligned}$$

$$8{:}\,400 = (7{:}\,103,\ 1{:}\,0) \quad 7{:}\,103 = (6{:}\,13,\ 1{:}\,0) \quad 6{:}\,13 = (1{:}\,0,\ 5{:}\,13)$$
$$5{:}\,13 = (4{:}\,4,\ 1{:}\,0) \quad 4{:}\,4 = (3{:}\,1,\ 1{:}\,0)$$

9.3.7 (a) We omit the pictures.

(b) In the notation introduced in Exercise 9.3.5, with $n = 2m+1$ and $k = b_n/2$, we claim that $\mathcal{M}_n = n{:}\,k = (m+1{:}\,0,\ m{:}\,0)$. To prove this, note that the rank of this tree is $b_1b_{2m} + \cdots + b_mb_{m+1}$ and that

$$b_n = b_1b_{2m} + \cdots + b_{2m}b_1 = 2(b_1b_{2m} + \cdots + b_mb_{m+1}).$$

(c) If $n = 2m$, then $b_n = 2(b_1b_{2m-1} + \cdots + b_{m-1}b_{m+1}) + b_m^2$, which is divisible by 2 if and only if b_m is. Thus there is no such tree unless b_m is even. In this case you should be able to show that $\mathcal{M}_{2m} = (m{:}\,b_m/2,\ m{:}\,0)$.

9.3.9. Here is one way to define an equivalence relation $\equiv$ by induction on the number of vertices. Let $\mathcal{T}_n$ be the set of labeled n-vertex RP-trees. Define all the trees in $\mathcal{T}_1$ to be equivalent. If $n > 1$, suppose $T, T' \in \mathcal{T}_n$. Let T be built from $T_1, \ldots, T_k$ and T' from $T'_l, \ldots, T'_\ell$ according to the recursive construction in Example 7.9 (p. 206). We define $T \equiv T'$ if and only $k = \ell$ and $T_i \equiv T'_i$ for $1 \le i \le k$.

9.3.11 (a) An x_i belongs in a parenthesis pair that has nothing inside it. Number the empty pairs from left to right and insert x_i into the ith pair.

(b) This is just a translation of what has been said. If you are confused, remember that $\mathtt{B}(n)$ should is to be thought of as all possible parentheses patterns for $x_1, \ldots, x_n$.

(c) This simply involves the replacement described in (b): Make $\bullet$ correspond to ( ) and make the tree with sons T_1 and T_2 correspond to (P_1P_2), where P_i is the parentheses pattern corresponding to T_i.

9.3.13 (a) The leaves in an RP-tree are distinguishable because the tree is ordered. Thus, each marking of the n leaves leads to a different situation. The same comments applies to vertices and there are $2n-1$ vertices by Exercise 9.3.6.

(b) Mark the single vertex that arises in this way to obtain an element of $\mathcal{V}_n$ Interchanging x and the tree rooted at b gives a different element of $\mathcal{L}_{n+1}$ that gives rise to the same element of $\mathcal{V}_n$.

Conversely, given any element of $\mathcal{V}_n$, the marked vertex should be split into two, f and b with b a son of f. Introduce another son x of f which is a marked leaf. There are two possibilities–make f a left son or a right son.

(c) By (a), $|\mathcal{L}_n| = nb_n$ and $|\mathcal{V}_n| = (2n-1)b_n$. By (b), $|\mathcal{L}_{n+1}| = 2|\mathcal{V}_n|$.

(d) By the recursion,

$$b_n \;=\; \frac{2(2n-3)}{n}b_{n-1} \;=\; \frac{2(2n-3)}{n}\frac{2(2n-5)}{n-1}b_{n-2} \;=\; \cdots \;=\; \frac{2^{n-1}(2n-3)(2n-5)\cdots 1}{n(n-1)\cdots 2}b_1.$$

Using $b_1 = 1$, we have a simple formula; however, it can be written more compactly:

$$\begin{aligned} b_n \;&=\; \frac{2^{n-1}(2n-3)(2n-5)\cdots 1}{n!} \;=\; \frac{2^{n-1}(n-1)!\,(2n-3)(2n-5)\cdots 1}{(n-1)!\,n!} \\ &=\; \frac{(2n-2)!}{(n-1)!\,n!} \;=\; \frac{1}{n}\binom{2n-2}{n-1}. \end{aligned}$$

Section 10.1

10.1.1. The p and q calculations can be done by multiplication. If so, and we are asked for the coefficient of x^3, say, then we can ignore any power of x greater than x^3 that appear in intermediate steps.

(a) Letting $\equiv$ mean equality of coefficients of x^n for $n \le 3$, we have

$$\begin{aligned} p^2 \;&=\; (1+x+x^2+x^3)^2 \;\equiv\; 1+2x+3x^2+4x^3 \\ p^3 \;&\equiv\; (1+x+x^2+x^3)(1+2x+3x^2+4x^3) \;\equiv\; 1+3x+6x^2+10x^3 \\ p^4 \;&\equiv\; (1+x+x^2+x^3)(1+3x+6x^2+10x^3) \;\equiv\; 1+4x^2+10x^3+20x^4. \end{aligned}$$

(b) By the opening remarks in the solution, this will be the same as (a).

(c) We can do this by writing $r \equiv 1 + x + x^2 + x^3$ or we can write, for example, $f(x) = (1-x)^{-2}$ and use Taylor's Theorem to compute the coefficients.

(d) Note that $r = 1 + x + x^2 + x^3 + \cdots$. Whenever you add, subtract or multiply power series and look for the coefficient of some power of x, say x^n, only those powers of x that do not exceed n in the original series matter. Each of p, q and r begin $1 + x + x^2 + x^3$.

10.1.3. We have

$$(x^2 + x^3 + x^4 + x^5 + x^6)^8 \;=\; x^{16}(1 + x + x^2 + x^3 + x^4)^8 \;=\; x^{16}\left(\frac{1-x^5}{1-x}\right)^8.$$

The coefficient of x^{21} in this is the coefficient of x^5 in the eighth power on the right hand side. Since $(1-x^5)^8 = 1 - 8x^5 + \cdots$, this is simply the coefficient of x^5 in $(1-x)^{-8}$ minus 8 times the coefficient of x^0 (the constant term) in $(1-x)^{-8}$. Thus our answer is

$$(-1)^5\binom{-8}{5} - 8 \;=\; \binom{12}{5} - 8 \;=\; 784.$$

10.1.5. We'll do just the general k.

(a) We have $x^k A(x) = \sum_{m\geq 0} a_m x^{m+k} = \sum_{n\geq k} a_{n-k}x^n$. Thus the coefficient of x^n is 0 for $n < k$ and a_{n-k} for $n \geq k$.

(b) We have

$$\left(\frac{d}{dx}\right)^k A(x) \;=\; \sum_{m=0}^{\infty} a_m \left(\frac{d}{dx}\right)^k x^m \;=\; \sum_{m=k}^{\infty} a_m (m)(m-1)\cdots(m-k+1)x^{m-k}.$$

Set $n = m - k$ to obtain the answer: $a_{n+k}(n+k)(n+k-1)\cdots(n+1) \;=\; a_{n+k}\frac{(n+k)!}{n!}$.

(c) Since $\left(x\frac{d}{dx}\right)A(x) = \sum_{m=0}^{\infty} m a_m x^m$, repeating the operation k times leads to $\sum_{m=0}^{\infty} m^k a_m x^m$. Thus the answer is $n^k a_n$.

10.1.7. This is simply the derivation of (10.4) with r used instead of 1/3. The generating function for the sum is $S(x) = 1/(1 - r(1+x))$ and the coefficient of x^k is

$$\frac{\Big(r/(1-r)\Big)^k}{1-r} \;=\; \frac{\Big(r/(1-r)\Big)^{k+1}}{r} \;=\; \frac{r^k}{(1-r)^{k+1}}.$$

To verify convergence, let $a_n = \binom{n}{k} r^n$ and note that

$$\lim_{n\to\infty}\left|\frac{a_{n+1}}{a_n}\right| \;=\; \lim_{n\to\infty}\frac{(n+1)|r|}{n-k+1} \;=\; |r| \;<\; 1.$$

10.1.9. This is very similar to the Exercise 10.1.8 With $a_j = (-1)^j\binom{m}{j}$ and $b_j = \binom{m}{j}$, we can apply the convolution formula. The result is $C(x) = (1-x)^m(1+x)^m = (1-x^2)^m$. By the binomial theorem, $(1-x^2)^m = \sum(-1)^j\binom{m}{j}x^{2j}$. Thus, the sum we are to simplify is zero if k is odd and $(-1)^j\binom{m}{j}$ if $k = 2j$.

10.1.11. The essential fact is that $\sum_{s=0}^{k-1}\omega^{rs}$ is k if r is multiple of k and 0 otherwise.

10.1.13. This is multisection with $k = 3$ and $j = 0, 2, 1$, respectively. The basic facts that are needed are $e^{i\theta} \;=\; \cos\theta + i\sin\theta$ and the sine and cosine of various angles in the 30°-60°-90° right triangle.

Section 10.2

10.2.1 (a) Let $a_n = 5a_{n-1} - 6a_{n-2} + b_n$ where $b_1 = 1$ and $b_n = 0$ for $n \neq 1$. Then

$$A(x) = \sum_{k=0}^{\infty}(5xa_{k-1}x^{k-1} - 6x^2a_{k-2}x^{k-2}) + x = 5xA(x) - 6x^2A(x) + x.$$

Thus

$$A(x) = \frac{x}{1-5x+6x^2} = \frac{1}{1-3x} - \frac{1}{1-2x}$$

and $a_n = 3^n - 2^n$.

(b) To correct the recursion, add c_{n+1} to the right side, where $c_0 = 1$ and $c_n = 0$ for $n \neq 0$. Multiply both sides by x^{n+1} and sum to obtain $A(x) = xA(x) + 6x^2A(x) + 1$. With some algebra,

$$A(x) = \frac{1}{1-x-6x^2} = \frac{3/5}{1-3x} + \frac{2/5}{1+2x}$$

and so $a_n = (3^{n+1} - (-2)^{n+1})/5$.

(c) To correct the recursion, add b_n where $b_1 = 1$ and $b_n = 0$ otherwise. Thus $A(x) = xA(x) + x^2A(x) + 2x^3A(x) + x$ and so

$$A(x) = \frac{x}{1-x-x^2-2x^3} = \frac{x}{(1-2x)(1+x+x^2)}.$$

By the quadratic formula, we can factor $1 + x + x^2$ as $(1 - \omega x)(1 - \overline{\omega}x)$, where $\omega = (-1 + \sqrt{-3})/2$ and $\overline{\omega}$ is the complex conjugate of ω. Using partial fractions,

$$A(x) = \frac{2/7}{1-2x} - \frac{(3-2\sqrt{-3})/21}{1-\omega x} - \frac{(3+2\sqrt{-3})/21}{1-\overline{\omega}x}$$

and so

$$a_n = \frac{2^{n+1}}{7} - \frac{(3-2\sqrt{-3})\omega^n}{21} - \frac{(3+2\sqrt{-3})\overline{\omega}^n}{21}.$$

The last two terms are messy, but they can be simplified considerably by noting that $\omega^3 = 1$ and so they are periodic with period 3. Thus

$$a_n = \frac{2^{n+1}}{7} + \begin{cases} (-2/7) & \text{if } n/3 \text{ has remainder 0;} \\ 3/7 & \text{if } n/3 \text{ has remainder 1;} \\ (-1/7) & \text{if } n/3 \text{ has remainder 2.} \end{cases}$$

(d) The recursion holds for $n = 0$ as well. From the recursion, $A(x) = 2xA(x) + \sum nx^n$. By Exercise 10.1.5, the sum is $x\frac{d}{dx}\sum x^n$, which is $x/(1-x)^2$. Thus

$$A(x) = \frac{x}{(1-x)^2(1-2x)} = \frac{2}{1-2x} - \frac{1}{1-x} - \frac{1}{(1-x)^2}.$$

After some algebra with these, we obtain $a_n = 2^{n+1} - n - 2$.

10.2.3. Start with a string of $n - i$ zeroes. Choose without repetition i of the $n + 1 - i$ positions (before all the zeroes or after any zero) and insert a one in each position chosen. The result is an n long string with i ones, none of them adjacent. The process is reversible: The position of a one is the number of zeroes preceding it. The formula for F_n follows immediately.

10.2.5 (a) Replacing A, B and C with their definitions and rearranging leads to

$$L_1L_2 + L_1H_2x^m + L_2H_1x^m + H_1H_2x^{2m} \;=\; (L_1 + H_1x^m)(L_2 + H_2x^m).$$

(b) The number of multiplications required by any procedure is an upper bound on $M(2m)$. There are three products of polynomials of degree m or less in our "less direct" procedure. If they are done as efficiently as possible, we will have $M(2m) \leq 3M(m)$.

(c) Let $s_k = M(2^k)$. We have $s_0 = 1$ and $s_k \leq 3s_{k-1}$ for $k > 0$. If we set $t_0 = 1$ and $t_k = 3t_{k-1}$ for $k > 0$, then $s_k \leq t_k$. The recursion gives $T(x) = 3xT(x) + 1$ and so $t_k = 3^k$. Thus, with $n = 2^k$, $M(n) \leq 3^k = (2^{\log_2 3})^k = n^{\log_2 3}$. From tables or a calculator, $\log_2 3 = 1.58\cdots$.

(d) To begin with, $L_1(x) = 1 + 2x$, $H_1(x) = -1 + 3x$, $L_2(x) = 5 + 2x$ and $H_2(x) = -x$. The product $L_1L_2 = (1 + 2x)(5 + 2x)$ is computed using the algorithm. The values are

$$m = 1, \quad A \;=\; (2)(2) \;=\; 4, \quad B \;=\; (1)(5) \;=\; 5 \quad \text{and} \quad C = (1 + 2)(5 + 2) = 21.$$

Thus $L_1L_2 \;=\; 5 + 12x + 4x^2$. In a similar way, the products $(-1 + 3x)(-x) \;=\; x - 3x^2$ and $(5x)(5 + x) \;=\; 25x + 5x^2$ are computed these are combined to give the final result:

$$(5 + 12x + 4x^2) + (x - 3x^2)x^4 + ((25x + 5x^2) - (5 + 12x + 4x^2) - (x - 3x^2))x^2,$$

which is $5 + 12x - x^2 + 12x^3 + 4x^4 + x^5 - 3x^6$.

(e) We'll just look at the case in which $n = 2m = 2^k$. Let a_k be the number of additions and subtractions needed. We have $a_0 = 0$ and, for $k > 0$, a_k equals $3a_{k-1}$ plus the number of additions and subtractions needed to prepare for and use the three multiplications. Preparation requires two additions of polynomials of degree $m - 1$. The results are three polynomials of degree $2m - 2$. We must perform two subtractions of such polynomials. Finally, the multiplication by x^m and x^{2m} arranges things so that there is some overlap among the coefficients. In fact, there will be $2m - 2$ additions required because of these overlaps (unless some coefficients happen to turn out zero). Since a polynomial of degree d has $d+1$ coefficients, there are a total of

$$2(m - 1 + 1) + 2(2m - 2 + 1) + (2m - 2) \;=\; 4n - 4.$$

Thus $a_k \;=\; 3a_{k-1}+4\times 2^k-4$ and so $A(x) \;=\; 3xA(x)+4\sum_{k>0}(2x)^k-4\sum_{k>0}x^k$. Consequently, $A(x) \;=\; 4x/(1 - x)(1 - 2x)(1 - 3x)$ and $a_k \;=\; 2 \times 3^{k+1} - 2^{k+3} + 2$. Comparing this with the multiplication result, we see that we need about three times as many additions and/or subtractions as we do multiplications, which is still much smaller than n^2 for large n.

10.2.7. When we use the initial conditions and solve for $A(x)$ we get $A(x) = R(x) + \frac{N(x)}{D(x)}$ where $R(x)$ is some polynomial, $D(x) = 1 - c_1x - \cdots - c_kx^k$ and $N(x)$ is a polynomial of degree less than k. By the Fundamental Theorem of Algebra, we can factor $D(x)$ as given in the exercise. By the theory of partial fractions, there are constants $b_{i,j}$ such that

$$\frac{N(x)}{D(x)} \;=\; \sum_{i=1}^{m}\sum_{j=1}^{d_i} b_{i,j}(1 - r_ix)^{-j}.$$

Equating coefficients an assuming n is larger than the degree of $R(x)$, we have

$$a_n \;=\; \sum_{i=1}^{m}\sum_{j=1}^{d_i} b_{i,j}\binom{-j}{n}(-r_i)^n \;=\; \sum_{i=1}^{m}\sum_{j=1}^{d_i} b_{i,j}\binom{n+j-1}{j-1}r_i^n.$$

Since $\binom{n_j-1}{j-1}$ is a polynomial in n of degree $j-1$, it follows that $\sum_{j=1}^{d_i} b_{i,j}\binom{n+j-1}{j-1}$ is a polynomial in n of degree at most $d_i - 1$.

Let d be the degree of $R(x)$, where the degree of 0 is $-\infty$. If ℓ is the largest value of n for which an initial value of n must be specified, then $d \le \ell - k$. To find the coefficients of the polynomials $P_i(n)$, it suffices to know the values of $a_t, \ldots, a_{t_k}$ for any t larger than the degree of $R(x)$.

Section 10.3

10.3.1 (a) $(1-x)D' - D = -e^{-x} = -(1-x)D$ and so $(1-x)D' - xD = 0$.

(b) The coefficient of x^n on the left of our equation in (a) is

$$\frac{D_{n+1}}{n!} - \frac{D_n}{(n-1)!} - \frac{D_{n-1}}{(n-1)!}.$$

The initial conditions are $D_0 = 1$ and $D_1 = 0$.

10.3.3 (a) We are asked to solve $Q'(x) - 2(1-x)^{-1}Q(x) = 2x(1-x)^{-3}$. The integrating factor is

$$\exp\left(\int -2(1-x)^{-1}dx\right) = \exp\big(2\ln(1-x)\big) = (1-x)^2.$$

Thus

$$Q(x)(1-x)^2 = \int \frac{2x}{1-x}dx = \int \left(-2 + \frac{2}{1-x}\right)dx = -2x - 2\ln(1-x) + C.$$

(b) We have $-2\ln(1-x) - 2x = \sum_{k\ge 2} 2x^k/k$ and

$$(1-x)^{-2} = \sum_{k\ge 0}\binom{-2}{k}(-x)^k = \sum_{k\ge 0}(k+1)x^k.$$

By the formula for the coefficients in a product of generating functions,

$$\begin{aligned} q_n &= \sum_{k=2}^{n}\frac{2(n-k+1)}{k} = 2(n+1)\sum_{k=2}^{n}\frac{1}{k} - \sum_{k=2}^{n}2 \\ &= 2(n+1)\sum_{k=1}^{n}\frac{1}{k} - 2(n+1) - 2(n-1) = 2(n+1)\sum_{k=1}^{n}\frac{1}{k} - 4n. \end{aligned}$$

Section 10.4

10.4.1 (a) This is nothing more than a special case of the Rule of Product—at each time we can choose anything from $\mathcal{T}$.

(b) Simply sum the previous result on k.

(c) The hint tells how to do it. All that is left is algebra.

(d) The solution is like that in the previous part, except that we start with

$$\prod_{T\in\mathcal{T}}\left(\sum_{i=0}^{\infty}\left(\mathbf{x}^{\mathbf{w}(T)}\right)^i\right) \;=\; \prod_{T\in\mathcal{T}}(1-\mathbf{x}^{\mathbf{w}(T)})^{-1}.$$

10.4.3 (a) This is simply $2^*\ \{0,\ 12^*\}^*$. Thus the generating function is

$$A(x) \;=\; \frac{1}{1-x}\,\frac{1}{1-\left(x+x\dfrac{1}{1-x}\right)} \;=\; \frac{1}{1-3x+x^2}.$$

Multiply both sides by $1-3x+x^2$ and equate coefficients of x^n to obtain the recursion

$$a_n \;=\; 3a_{n-1}-a_{n-2} \quad \text{for} \quad n>1$$

with initial conditions $a_0=1$ and $a_1=3$.

(b) You should be able to see that this is described by $0^*(11^*0^k0^*)^*$. Since

$$\mathsf{G}_{11^*0^k0^*} \;=\; x\frac{1}{1-x}x^k\frac{1}{1-x} \;=\; \frac{x^{k+1}}{(1-x)^2},$$

the generating function we want is

$$A(x) \;=\; \frac{1}{1-x}\,\frac{1}{1-x^{k+1}/(1-x)^2} \;=\; \frac{1-x}{1-2x+x^2-x^{k+1}}.$$

Clearing of fractions and equating coefficients, we obtain the recursion

$$a_n \;=\; 2a_{n-1}-a_{n-2}+a_{n-k-1} \quad \text{for} \quad n>1,$$

with the understanding that $a_j=0$ for $j<0$. The initial conditions are $a_0=a_1=1$.

(c) A possible formulation is

$$0^*\ (1(11)^*00^*)^*\ \{\lambda,\ 1(11)^*\}.$$

This says, start with any number of zeroes, then append any number of copies of the patterns of type Z (described soon) and then follow by either nothing or an odd number of ones. A pattern of type Z is an odd number of ones followed by one or more zeroes. The translation to a generating function gives

$$\frac{1}{1-x}\frac{1}{\mathsf{G}_Z(x)}\left(1+x\frac{1}{1-x^2}\right) \quad \text{where} \quad \mathsf{G}_Z(x) \;=\; x\frac{1}{1-x^2}x\frac{1}{1-x}.$$

After some algebra, the generating function reduces to

$$A(x) \;=\; \frac{1+x-x^2}{1-x-2x^2+x^3},$$

which gives $a_n \;=\; a_{n-1}+2a_{n-2}-a_{n-3}$ for $n>2$, with initial conditions $a_0=1$, $a_1=2$ and $a_2=3$.

10.4.5. Here's a way to construct a pile of height h. Look at the number of blocks in each column. The numbers increase to h, possibly stay at h for some time, and then fall off. The numbers up to but not including the first h form a partition of a number with largest part at most $h-1$ and the numbers after the first h form a partition of a number with largest part at most h. The structures are these partitions. By the Rule of Product and Exercise 10.4.4

$$\sum_{n\geq 0} s_{n,h}(x) \;=\; \Big(\prod_{i=1}^{h-1} \frac{1}{1-x_i}\Big)\, x^h \,\Big(\prod_{i=1}^{h} \frac{1}{1-x_i}\Big) \;=\; \frac{x^h}{(1-x^h)\displaystyle\prod_{i=1}^{h-1}(1-x^i)^2}.$$

Summing this over all $h>0$ and adding 1 gives $\sum s_n x^n$. No simple formula is known for the sum.

10.4.7 (a) We can build the trees by taking a root and joining to it zero, one or two binary RP-trees. This gives us $T(x) = x(1+T(x)+T(x)^2)$.

(b) There is no simple expansion for the square root; however, various things can be done. One possibility is to use $\sqrt{1-2x-3x^2} = \sqrt{1-3x}\,\sqrt{1+x}$. You can then expand each square root and multiply the generating functions together. This leads to a summation of about n terms for p_n. The terms alternate in sign. A better approach is to write

$$\sqrt{1-2x-3x^2} \;=\; \sum_k \binom{1/2}{k}(-1)^k(2x+3x^2)^k \;=\; \sum_{k,j}(-1)^k\binom{1/2}{k}\binom{k}{j}2^{k-j}3^j x^{k+j}.$$

This leads to a summation of about $n/2$ positive terms for p_n. It's also possible to get a recursion by constructing a first order, linear differential equation with polynomial coefficients for $T(x)$ as done in Exercise 10.2.6. Since the recursion contains only two terms, it's the best approach if we want to compute a table of values. It's also the easiest to program on a computer.

10.4.9. The key to working this problem is to never allow the root to have exactly one son.

(a) Let the number be r_n. The generating function for those trees whose root has degree k is $R(x)^k$. Since $\sum_{k\geq 0} R(x)^k = 1/(1-R(x))$, we have $R(x) = x\frac{1}{1-R(x)} - xR(x)$. Clearing of fractions and solving the quadratic,

$$R(x) \;=\; \frac{1+x-\sqrt{1-2x-3x^2}}{2(1+x)}.$$

(The minus sign is the correct choice for the square root because $R(0) = r_0 = 0$.) These numbers are closely related to p_n in Exercise 10.4.7. By comparing the equations for the generating functions,

$$(1+x)R(x) \;=\; x(P(x)+1)$$

and so $r_n + r_{n-1} = p_{n-1}$ when $n>1$.

(b) We modify the previous idea to count by leaves:

$$R(x) \;=\; x + \sum_{k\geq 2} R(x)^k \;=\; x + \frac{R(x)^2}{1-R(x)}.$$

Solving the quadratic:

$$R(x) \;=\; \frac{1+x-\sqrt{1-6x+x^2}}{4}.$$

(c) From (a) we have $2(1+x)R - 1 - x = -\sqrt{1-2x-3x^2}$ and so

$$2xR' + 2R - 1 = \frac{1-3x}{\sqrt{1-2x-3x^2}}.$$

Thus $(1-2x-3x^2)(2xR'+R-1) = -2(1+x)R+1+x$. Equating coefficients of x^n gives us

$$(2n+1)r_n - (4n-2)r_{n-1} - (6n-9)r_{n-2} = -2r_n - 2r_{n-1} \quad \text{for} \quad n \geq 3.$$

Rearranging and checking initial conditions we have

$$r_{n+1} = \frac{4nr_n + 3(2n-1)r_{n-1}}{2n+5} \quad \text{for} \quad n \geq 2,$$

with $r_0 = r_2 = 0$ and $r_1 = 1$. You should be able to treat (b) in a similar manner. The result is $r_0 = 0$, $r_1 = r_2 = 1$ and, for $n \geq 2$,

$$r_{n+1} = \frac{3(2n-1)r_n - (n-2)r_{n-1}}{n+1}.$$

10.4.11 (a) A tree of outdegree D, consists of a root and some number $d \in D$ of trees of outdegree D joined to the root. Use the Rules of Sum and Product.

(b) Let $f_0(x) = 1$. Define $f_{n+1}(x) = x\sum_{d\in D} f_n(x)^d$. We leave it to you to prove by induction that $f_n(x)$ agrees with $T_D(x)$ through terms of degree n.

(c) Except for the $1 \in D$ question, this is handled as we did $T_D(x)$. Why must we have $1 \notin D$? You should be able to see that there are an infinite number of trees with exactly one leaf—construct a tree that is just a path of length n from the root to the leaf.

10.4.13. We can build these trees up the way we built the full binary RP-trees: join two trees at a root. If we distinguish right and left sons, every case will be counted twice, except when the two sons are the same. Thus $B(x) = x + \frac{1}{2}(B(x)^2 - E(x)) + E(x)$, where $E(x)$ counts the situation where both sons are the same and nonempty. We get this by choosing a son and then duplicating it. Thus each leaf in the son is replaced by two leaves and so $E(x) = B(x^2)$.

10.4.15 (a) Either the list consists of repeats of just one item OR it consists of a list of the proper form AND a list of repeats of one item. In the first case we can choose the item in s ways and use it any number of times from 1 to k. In the second case, we can choose the final repeating item in only $s-1$ ways since it must differ from the item preceding it.

(b) After a bit of algebra,

$$A_k(x) = \frac{s/(s-1)}{1-(s-1)(x+x^2+\cdots+x^k)} - \frac{s}{s-1} = \frac{s(1-x)/(s-1)}{1-sx+(s-1)x^{k+1}} - \frac{s}{s-1}.$$

(c) Multiplying both sides of the formula just obtained for $A_k(x)$ by $1-sx+(s-1)x^{k+1}$ gives the desired result.

(d) Call a sequence of the desired sort acceptable. Add anything to the end of an n-long acceptable sequence. This gives $sa_{n,k}$ sequences. Each of these is either an acceptable sequence of length $n+1$ or an $(n-k)$-long acceptable sequence followed by $k+1$ copies of something different from the last entry in the $(n-k)$-long sequence.

10.4.17. We have $1 - 3x + x^2 = (1 - ax)(1 - bx)$ where $a = \frac{3+\sqrt{5}}{2}$ and $b = \frac{3-\sqrt{5}}{2}$. Thus

$$\frac{x}{1 - 3x + x^2} = \frac{1/(a-b)}{1 - ax} - \frac{1/(a-b)}{1 - bx}$$

and so, since $a - b = \sqrt{5}$,

$$r_n = \frac{a^n - b^n}{\sqrt{5}}.$$

10.4.19 (a) The accepting states are unchanged except that if the old start state was accepting, both the old and new start states are accepting. If there was an edge from the old start state to state t labeled with input i, then add an edge from the new start state to t labeled with i. (The old edge is *not* removed.) We can express this in terms of the map $f : S \times I \to 2^S$ for the nondeterministic automaton. Let $s_o \in S$ be the old start state and introduce a new start state s_n. Let $T = S \cup \{s_n\}$ and define $f^* : T \times I \to 2^T$ by

$$f*(t,i) = \begin{cases} f(t,i), & \text{if } t \in S, \\ f(s_o,i), & \text{if } t = s_n. \end{cases}$$

(b) Label the states of $\mathcal{A}$ and $\mathcal{B}$ so that they have no labels in common. Call their start states s_A and s_B. Add a new start state s_n that has edges to all of the states that s_A and s_B did. In other words, $f*(s_n, i)$ is the union of $f_A(s_A, i)$ and $f_B(s_B, i)$, where f_A and f_B are the functions for $\mathcal{A}$ and $\mathcal{B}$. If either s_A or s_B was an accepting state, so is s_n; otherwise the accepting states are unchanged.

(c) Add the start state of $S(\mathcal{A})$ to the accepting states. (This allows the machine to accept the empty string, which is needed since * means "*zero* or more times.") Run edges from the accepting states of $S(\mathcal{A})$ to those states that the start state of $S(\mathcal{A})$ goes to. In other words, if s is the start state,

$$f^*(t,i) = \begin{cases} f(t,i), & \text{if } t \text{ is not an accepting state,} \\ f(t,i) \cup f(s,i), & \text{if } t \text{ is an accepting state.} \end{cases}$$

(d) From each accepting state of $\mathcal{A}$, run an edge to each state to which the start state of $\mathcal{B}$ has an edge. The accepting states of $\mathcal{B}$ are accepting states. If the start state of $\mathcal{B}$ is an accepting state, then the accepting states of $\mathcal{A}$ are also accepting states, otherwise they are not. The start state is the start state of $\mathcal{A}$.

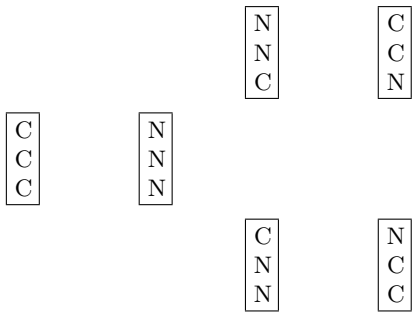


Figure S.11.1 The state transition digraph for covering a 3 by n board with dominoes. Each vertex is labeled with a triple that indicates whether commitment has been made in that row (C) or not made (N). The start and end states are those with no commitments.

Section 11.1

11.1.1. The problem is to eliminate all but the c's from the recursion. One can develop a systematic method for doing this, but we will not since we have generating functions at our disposal. In this particular case, let $p_n = f_n + s_n$ and note that $p_n - p_{n-1} = 2c_{n-1}$ by (11.5). Thus, by the first of (11.4), this result and the last of (11.5),

$$\begin{aligned} c_{n+1} - c_n &= (2c_n + p_n + c_{n-1}) - (2c_{n-1} + p_{n-1} + c_{n-2}) \\ &= 2c_n - c_{n-1} - c_{n-2} + (p_n - p_{n-1}) \\ &= 2c_n + c_{n-1} - c_{n-2}. \end{aligned}$$

11.1.3 (a) Figure S.11.1 gives a state transition digraph.

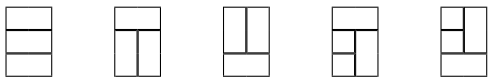
Let $a_{n,s}$ be the number of ways to take n steps from the state with no commitments and end in a state s. Let $A_s(x) = \sum_n a_{n,s}x^n$. As in the text, the graph lets us write down the linked equations for the generating functions. From the graph it can be seen that A_s depends only on the number k of commitments in s. Therefore we can write $A_s = B_k$. The linked equations are then

$$\begin{aligned} B_0(x) &= x(B_3(x) + 2B_1(x)) + 1 \\ B_1(x) &= x(B_0(x) + B_2(x)) \\ B_2(x) &= xB_1(x) \\ B_3(x) &= xB_0(x), \end{aligned}$$

which can be solved fairly easily for $B_0(x)$.

(b) Equate coefficients of x^n on both sides of $(1 - 4x^2 + x^4)A(x) = 1 - x^2$.

(c) By looking at the dominoes in the last two columns of a board, we see that it can end in five mutually exclusive ways:



This shows that a_n equals $3a_{n-2}$ plus whatever is counted by the last two of the five cases. A board of length $n-2$ ends with either (i) one vertical domino and one horizontal dominoes or (ii) three horizontal dominoes. If the vertical dominoes mentioned in (i) are changed to the left ends of horizontal dominoes, they fit with the last two cases shown above. If the three horizontal mentioned in (ii) are removed, we obtain all boards of length $n-4$. Thus the sum of the last two cases in the picture plus a_{n-4} equals a_{n-2}.

11.1.5. Call the start state α and let $L_{i,j}$ be the number of different single letter inputs that allow the machine to move from state i to state j. Let $a_{n,i}$ be the number of ways to begin in state α, recognize n letters and end in state i and let $A_i = \mathsf{G}^1(a_{n,i})$. The desired generating function is the sum of A_i over all accepting states. A linked set of recursions can be obtained from the automaton that leads to the generating function equations

$$A_i(x) \;=\; \sum_j L_{i,j} x A_j(x) + \begin{cases} 1 & \text{if } i = \alpha; \\ 0 & \text{otherwise.} \end{cases}$$

11.1.7 (a) We will use induction. It is true for $n = 1$ by the definition of $m_{x,y} = m^{(1)}_{x,y}$. (Its also true for $n = 0$ because the zeroth power of a matrix is the identity and so $m^{(0)}_{x,y} = 1$ if $x = y$ and 0 otherwise.) Now suppose that $n > 1$. By the definition of matrix multiplication, $m^{(n)}_{x,y} = \sum_z m^{(n-1)}_{x,z} m_{z,y}$. By the induction hypothesis and the definition of $m_{z,y}$ each term in the sum is the number of ways to get from x to z in $n-1$ steps times the number of ways to get from z to y in one step. By the Rules of Sum and Product, the proof is complete.

(b) If α is the initial state, $\mathbf{i}M^n\mathbf{a}^t = \sum m^{(n)}_{\alpha,y}$, the sum ranging over all accepting states y.

(c) By the previous part, the desired generating function is

$$\sum_{n=0}^{\infty} \mathbf{i}M^n\mathbf{a}^t\, x^n \;=\; \mathbf{i}\sum_{n=0}^{\infty} x^n M^n \mathbf{a}^t \;=\; \mathbf{i}\sum_{n=0}^{\infty} (xM)^n \mathbf{a}^t \;=\; \mathbf{i}(I - xM)^{-1}\mathbf{a}^t.$$

(d) The matrix M is replaced by the table given in the solution to the previous exercise.

Section 11.2

11.2.1. Theorem. Suppose each structure in a set $\mathcal{T}$ of structures can be constructed from an ordered partition (K_1, K_2) of the labels, two nonnegative integers ℓ_1 and ℓ_2, and some ordered pair (T_1, T_2) of structures using the labels K_1 in T_1 and K_2 in T_2 such that:

(i) The number of ways to choose a T_i with labels K_i and ℓ_i unlabeled parts depends only on i, $|K_i|$ and ℓ_i.

(ii) Each structure $T \in \mathcal{T}$ arises in exactly one way in this process.

(We allow the possibility of $K_i = \emptyset$ if $\mathcal{T}_i$ contains structures with no labels and likewise for $\ell_i = 0$.) It then follows that

$$T(x, y) \;=\; T_1(x, y)T_2(x, y),$$

where $T_i(x, y) \;=\; \sum_{n=0}^{\infty} t_{i,n,m}(x^n/n!)y^m$ and $t_{i,n,m}$ is the number of ways to choose T_i with labels $\underline{n}$ and k unlabeled parts. Define $T(x, y)$ similarly.

The proof is the same as that for the original Rule of Product except that there is a double sum:

$$t_{n,m} \;=\; \sum_{K_1 \subseteq \underline{n}} \sum_{\ell_1=0}^{m} t_{1,|K_1|,\ell_1}\, t_{2,n-|K_1|,m-\ell_1} \;=\; \sum_{k=0}^{n} \sum_{\ell_1=0}^{m} \binom{n}{k} t_{1,k,\ell_1}\, t_{2,n-k,m-\ell_1}$$

11.2.3 (a) By the text,

$$\sum_k z(n, k)y^k \;=\; y(y + 1) \cdots (y + n - 1).$$

Replacing all but the last factor on the right hand side gives us

$$\sum_k z(n,k)y^k \;=\; \Big(\sum_k z(n-1,k)y^k\Big)(y+n-1).$$

Equate coefficients of y^k.

(b) For each permutation counted by $z(n,k)$, look at the location of n. There are $z(n-1,k-1)$ ways to construct permutations with n in a cycle by itself. To construct a permutation with n not in a cycle by itself, first construct one of the permutations counted by $z(n-1,k)$ AND then insert n into a cycle. Since there are j ways to insert a number into a j-cycle, the number of ways to insert n is the sum of the cycle lengths, which is $n-1$.

11.2.5 (a) For any particular letter appearing an odd number of times, the generating function is

$$\sum_{n \text{ odd}} \frac{x^n}{n!} \;=\; \frac{e^x - e^{-1}}{2} \qquad \text{with Taylor's theorem and some work.}$$

We must add 1 to this to allow for the letter not being used. The Rule of Product is then used to combine the results for A, B and C.

(b) Multiplying out the previous result:

$$\begin{aligned}\left(1+\frac{e^x-e^{-x}}{2}\right)^3 &= 1+3(e^x-e^{-x})/2+3(e^x-e^{-x})^2/4+(e^x-e^{-x})^3/8\\ &= 1+\Big(3e^x/2-3e^{-x}/2\Big)+\Big(3e^{2x}/4-3/2+3e^{-2x}/4\Big)+\Big(e^{3x}/8-3e^x/8+3e^{-x}/8-e^{-3x}/8\Big)\\ &= -1/2+\Big(e^{3x}/8-e^{-3x}/8\Big)+\Big(3e^{2x}/4+e^{-2x}/4\Big)+\Big(9e^x/8-9e^{-x}/8\Big).\end{aligned}$$

Now compute the coefficients.

11.2.7. We saw in this section that $B(x) \;=\; \exp(e^x-1)$. Differentiating:

$$B'(x) \;=\; \exp(e^x-1)\,(e^x-1)' \;=\; B(x)e^x.$$

Equating coefficients of x^n:

$$\frac{B_{n+1}}{n!} \;=\; \sum_{k=0}^{n} \frac{B_k}{k!}\,\frac{1}{(n-k)!},$$

which gives the result.

11.2.9 (a) Let $g_{n,k}$ be the number of graphs with n vertices and k components. We have $\sum_{n,k} g_{n,k}(x^n/n!)y^k = \exp(yC(x))$, by the Exponential Formula. Differentiating with respect to y and setting $y=1$ gives us

$$\sum_n \Big(\textstyle\sum_k k g_{n,k}\Big)x^n/n! \;=\; \dfrac{\partial \exp(yC(x))}{\partial y}\bigg|_{y=1} \;=\; H(x).$$

(b) $\sum_k g_{n,k}$ is the number of ways to choose an n-vertex graph and mark a component of it. We can construct a graph with a marked component by selecting a component (giving $C(x)$) AND selecting a graph (giving $G(x)$).

(c) For permutations, there are $(n-1)!$ connected components of size n and so

$$C(x) \;=\; \sum \frac{(n-1)!x^n}{n!} \;=\; \sum x^n/n \;=\; -\ln(1-x).$$

Since there are $n!$ permutations of n, $G(x) = \frac{1}{1-x}$ and the average number of cycles in a permutation is

$$\frac{h_n}{n!} \;=\; n!\,[x^n] \sum_{k=1}^{\infty} \frac{x^n}{n} \;\frac{1}{1-x} \sum_{k=1}^{n} \frac{1}{k}.$$

(d) Since $C(x) = e^x - 1$, we have $H(x) = (e^x-1)\exp(e^x-1)$ and so

$$h_n \;=\; \sum_{k=1}^{n} \binom{n}{k} B_{n-k} \;=\; \sum_{k=0}^{n} \binom{n}{k} B_{n-k} - B_n,$$

which is $B(n+1) - B_n$ by the previous exercise.

(e) Since $C(x) = x + x^2/2$, we have $H(x) = (x+x^2/2)I(X)$, where $I(x)$ is the EGF for i_n, the number of involutions of $\underline{n}$. Thus the average number of cycles in an involution of $\underline{n}$ is

$$\frac{\binom{n}{1} i_{n-1} + \binom{n}{2} i_{n-2}}{i_n} = \frac{n}{2}\left(1 + \frac{i_{n-1}}{i_n}\right),$$

where the right side comes from the recursion $i_n = i_{n-1} + (n-1)i_{n-2}$.

11.2.11. Suppose $n > 1$. Since f is alternating,

- k is even;
- $f(1), \ldots, f(k-1)$ is an alternating permutation of $\{f(1), \ldots, f(k-1)\}$;
- $f(k+1), \ldots, f(n)$ is an alternating permutation of $\{f(k+1), \ldots, f(n)\}$.

Thus, an alternating permutation of $\underline{n}$ for $n > 1$ is built from an alternating permutation of odd length AND an alternating permutation, such that the sum of the lengths is $n-1$. We have shown that

$$\sum_{n>1} \frac{a_n x^{n-1}}{(n-1)!} \;=\; B(x)A(x)$$

and so $A'(x) = B(x)A(x) + 1$. Similarly, $B'(x) = B(x)B(x) + 1$.

Separate variables in $B' = B^2 + 1$ and use $B(0) = 0$ to obtain $B(x) = \tan x$. Use the integrating factor $\cos x$ for

$$A'(x) = (\tan x)A(x) + 1$$

and the initial condition $A(0) = 1$ to obtain $A(x) = \tan x + \sec x$.

11.2.13 (a) The square of a k-cycle is

- another cycle of length k if k is odd;
- two cycles of length $k/2$ if k is even.

Using this, we see that the condition is necessary. With further study, you should be able to see how to take a square root of such a permutation.

(b) This is simply putting together cycles of various lengths using (a) and recalling that there are $(k-1)!$ k-cycles.

(c) By bisection $\displaystyle\sum_{\substack{k=1 \\ k \text{ odd}}}^{\infty} \frac{x^k}{k} = \tfrac{1}{2}\Big(\{-\ln(1-x)\} - \{-\ln(1-(-x))\}\Big)$.

(d) We don't know of an easier method.

11.2.15 (a) We'll give two methods. First, we use the Exponential Formula approach in Example 11.14. When we add a root, the number of leaves does not change—except when we started with nothing and ended up with a single vertex tree. Correcting for this exception gives us the formula.

Without the use of the Exponential Formula, we could partition the trees according to the degree k of the root, treating $k = 0$ specially because in this case the root is a leaf:

$$L(x, y) = xy + \sum_{k=1}^{\infty} L(x, y)^k / k!.$$

(b) This type of problem was discussed in Section 10.3. Recall that there are n^{n-1} n-vertex rooted labeled trees.

(c) Differentiate the equation in (a) with respect to y and set $y = 1$ to obtain

$$U(x) = xe^{T(x)}U(x) + x = T(x)U(x) + x,$$

where we have used the fact that $L(x, 1) = T(x) = xe^{T(x)}$. Solving for U: $U = \frac{x}{1-T}$. Differentiating $T(x) = xe^{T(x)}$ and solving for $T'(x)$ gives us $T' = \frac{T}{x(1-T)}$. Thus $x^2T' + x = \frac{x}{1-T}$, which gives the equation for $U(x)$.

We know that $t_n = n^{n-1}$. It follows from the equation for $U(x)$ that

$$\frac{u_n}{n!} = \frac{(n-1)^{n-2}}{(n-2)!}.$$

Thus $u_n/t_n = n/(1+x)^{1/x}$, where $x = \frac{1}{n-1}$. As $n \to \infty$, $x \to 0$ and, by l'Hôpital's Rule

$$(1+x)^{1/x} = \exp\left(\frac{\ln(1+x)}{x}\right) \to \exp(1) = e.$$

11.2.17 (a) There are several steps

- Since g is a function, each vertex of $\varphi(g)$ has outdegree 1. Thus the image of $\underline{n}^{\underline{n}}$ lies in $\mathcal{F}_n$.
- φ is an injection: if $\varphi(g) = \varphi(h)$, then $(x, g(x)) = (x, h(x))$ for all $x \in \underline{n}$ and so $g = h$.
- Finally φ is onto $\mathcal{F}_n$: If $(V, E) \in \mathcal{F}_n$, for each $x \in \underline{n}$ there is an edge $(x, y) \in E$. Define $g(x) = y$.

(b) Let's think in terms of a function g corresponding to the digraph. Let $k \in \underline{n}$. If the equation $g^t(k)$ has a solution, then k is on a cycle and will be the root of a tree. The other vertices of the tree are those $j \in \underline{n}$ for which $g^s(j) = k$ for some s.

(c) This is simply an application of Exercise 11.2.2.

(d) In the notation of Theorem 11.6, $T(x)$ is $T(x)$, $f(t) = e^T$ and $g(u) = -\ln(1-u)$. Thus $n(f_n/n!)$ is the coefficient of u^n in $e^{nu}(1-u)^{-1}$. Using the convolution formula for the coefficient of a product of power series, we obtain the result.

Section 11.3

11.3.1 (a) $AB = CD$ follows from $a_i + b_i = \min(a_i, b_i) + \max(a_i, b_i)$. C divides A if and only if $c_i \le a_i$ for all i. Thus C divides both A and B if and only if $c_i \le \min(a_i, b_i)$ for all i. Thus $C = \gcd(A, B)$. The claim that $D = \operatorname{lcm}(A, B)$ follows similarly.

(b) It follows from the definition of gcd that $\gcd(n, i)$ must divide both n and i. This completes the first part. From (a), we have $\gcd(ab, ac) = a\gcd(b, c)$. Apply this with $ab = n$ and $a = k$: $\gcd(n, i) = k\gcd(n, i/k)$.

(c) By letting $j = n/k$, we can see that the two forms of the sum are equivalent, so we need only prove one. Let g generate C_n, that is $g(i) = i + 1$ modulo n. Let $h = g^t$ so that $h(i) = i + t$ modulo n. Thus $h^k(i) = i + kt$. The smallest $k > 0$ such that h^k is the identity is thus the smallest k such that kt is a multiple of n. Also, the cycle form of h consists of k-cycles. Since there are n elements, there must be n/k k-cycles. Thus h contributes $z_k^{n/k}$ to the sum. We need to know how many values of t give a particular value of k. By looking at prime factorization, you should be able to see that $k\gcd(n, t) = n$ and so $\gcd(n, t) = n/k$. We can now use (b) with k replaced by n/k to conclude that the number of such t is $\varphi(n/(n/k)) = \varphi(k)$.

11.3.3 (a) A regular octahedron can be surrounded by a cube so that each vertex of the octahedron is the center of a face of the cube. The center of the octahedron is the center of the cube. A line segment from the center of the cube to a vertex of the cube passes through the center of the corresponding face of the octahedron. A line segment from the center of the cube to the center of an edge of the cube passes through the center of the corresponding edge of the octahedron.

(b) By the above correspondence, the answer will be the same as the symmetries of the cube acting on the faces of the cube. See (11.31).

(c) By the above correspondence it is the same as the answer for the edges of the cube. See the previous exercise.

11.3.5. The group is usually called S_4. Here are its $4! = 24$ elements:

- The identity, which gives x_1^4.
- $(4-1)! = 6$ elements which are 4-cycles, which give $6x_4$.
- $\binom{4}{2} = 6$ elements which consist of two 1-cycles and a 2-cycle, giving $6x_1^2x_2$.
- $\binom{4}{1)\times(3-1)!=8}$ elements which consist of a 1-cycle and a 3-cycle, giving $8x_1x_3$.
- $\frac{1}{2}\binom{4}{2} = 3$ elements which consist of two 2-cycles, giving $3x_2^2$.

Thus

$$Z_{S_n} = \frac{x_1^4 + 6x_4 + 6x_1^2x_2 + 8x_1x_3 + 2x_2^2}{24}.$$

Now apply Theorem 11.9:

$$\begin{aligned} x_1^4 &\implies x_1^{16} \\ x_4 &\implies x_4^4 \\ x_1^2x_2 &\implies x_1^4x_2^{2+2+2} = x_1^4x_2^6 \\ x_1x_3 &\implies x_1x_3^{1+1+3} = x_1x_3^5 \\ x_2^2 &\implies x_2^8 \end{aligned}$$

11.3.7. If the vertices belong to different cycles of length i and $j > i$ we get $x_{\operatorname{lcm}(i,j)}^{\gcd(i,j)}$ as in the digraph case and so we get

$$\prod_{i<j} \left(x_{\operatorname{lcm}(i,j)}\right)^{\nu_i\nu_j \gcd(i,j)}.$$

If the two cycles have the same length, we must be careful not to overcount because the edge is not directed. When the two vertices are in different cycles of length i we get x_i^i and there are $\binom{\nu_i}{2}$ such pairs of cycles. When the two vertices belong to the same cycle of length i, we must be extra careful: If the separation between the vertices on the cycle is $i/2$, the edge $\{u, v\}$ comes back after $i/2$ steps around the cycle as $\{v, u\}$, so to speak. Otherwise, it must go i steps around the cycle. Thus we get a contribution of either

$$x_i^{(i-1)\nu_i/2} \text{ for odd } i, \qquad x_i^{(i-2)\nu_i/2} x_{i/2}^{\nu_i} \text{ for even } i.$$

Putting this all together, $\prod x_i^{\nu_i}$ becomes

$$\Bigl(\prod_{i<j} \bigl(x_{\operatorname{lcm}(i,j)}\bigr)^{\nu_i \nu_j \gcd(i,j)}\Bigr)\Bigl(\prod_i x_i^{i\nu_i(\nu_i-1)/2}\Bigr)\Bigl(\prod_{i \text{ odd}} x_i^{(i-1)\nu_i/2}\Bigr)\Bigl(\prod_{i \text{ even}} x_i^{(i-2)\nu_i/2} x_{i/2}^{\nu_i}\Bigr).$$

Section 11.4

11.4.1 (a) We have $r^2 = 2r + 1$ and so $r = 1 + \sqrt{2}$ and $m = 1$. By the principle, we expect there is some constant A such that $a_n \sim A(1 + \sqrt{2})^n$.

(b) Since $A(x) = \frac{1+x}{1-2x-x^2}$, we have $p(x) = 1 + x$, $q(x) = 1 - 2x - x^2$, $r = \sqrt{2} - 1 = 1/(1 + \sqrt{2}\,)$ and $q'(r) = -2\sqrt{2}$. Thus $k = 1$ and we have

$$a_n \sim \frac{(-1)^1 \sqrt{2}\, n^{1-1}}{-2\sqrt{2}\, r^{n+1}} \;=\; \tfrac{1}{2}\bigl(1 + \sqrt{2}\bigr)^{n+1}.$$

(c) We have $1 - 2x - x^2 \;=\; (1 - ax)(1 - bx)$ where $a = 1 + \sqrt{2}$ and $b = 1 - \sqrt{2}$. Expanding by partial fractions:

$$\begin{aligned}
\frac{1+x}{1-2x-x^2} &= \frac{1}{1-2x-x^2} + \frac{x}{1-2x-x^2} \\
&= \frac{\frac{a}{a-b}}{1-ax} - \frac{\frac{b}{a-b}}{1-bx} \\
&\quad + \frac{\frac{1}{a-b}}{1-ax} - \frac{\frac{1}{a-b}}{1-bx} \\
&= \frac{(2+\sqrt{2}\,)/(2\sqrt{2}\,)}{1-ax} - \frac{(2-\sqrt{2}\,)/(2\sqrt{2}\,)}{1-bx}.
\end{aligned}$$

Thus $a_n \;=\; \frac{1}{2}(1 + \sqrt{2}\,)^{n+1} + \frac{1}{2}(1 - \sqrt{2}\,)^{n+1}$.

11.4.3. From the discussion in the example, you can see that merging two lists of lengths i and $j > i$ takes at least i comparison. Thus the example shows that the number of comparisons for merge sorting satisfies $T_n \;=\; f(n) + T(m) + T(n - m)$ where $m = \lfloor n/2 \rfloor$ and $m \le f(n) < n$. Apply Principle 11.3.

11.4.5. We'll use Principle 11.4 (p. 345) so $t_{n,k}$ will denote the kth term of the sum we're given.

(a) Since

$$\frac{t_{n,k+1}}{t_{n,k}} \;=\; \frac{n-k}{n}$$

is less than 1 and is close to 1 when k/n is small, we'll use Principle 11.5 (p. 346). Since

$$\frac{1-r_k}{k} \;=\; \frac{1-(n-k)/n}{k} = \frac{1}{n},$$

(11.38) gives the estimate

$$\sqrt{\frac{\pi n}{2}}\,\frac{n!\,n^{n+1}}{n!}.$$

(b) This is a bit more complicated than (a) since

$$\frac{t_{n,k+1}}{t_{n,k}} \;=\; \frac{k+1}{k}\,\frac{n-k}{n}$$

is greater than 1 for small k and less than 1 for k near n. Thus $t_{n,k}$ achieves its maximum somewhere between 1 and n, namely, when the above ratio equals 1. This leads to a quadratic equation for k which has the solution

$$k \;=\; \frac{-1+\sqrt{1+4n}}{2}.$$

Since this differs from $\sqrt{n}$ by at most a constant, we'll split the sum into two pieces at $k=\sqrt{n}$ and use Principle 11.5 (p. 346) for each half. Since each half has the same estimate, we simply double one result. Ignoring the fact that $\sqrt{n}$ is not an integer, we set $k=\sqrt{n}+j$ and use $j\ge 0$ as the new index of summation. Call the new terms $t'_{n,j}$. We have

$$\begin{aligned}
r_j \;&=\; \frac{t'_{n,j+1}}{t'_{n,j}} = \frac{t_{n,k+1}}{t_{n,k}} \;=\; \frac{k+1}{k}\,\frac{n-k}{n}\\
&=\; \frac{\sqrt{n}+j+1}{\sqrt{n}+j}\,\frac{n-\sqrt{n}-j}{n}\\
&=\; \left(1+\frac{1}{\sqrt{n}+j}\right)\left(1-\frac{\sqrt{n}+j}{n}\right)\\
&=\; 1+\frac{n-\big(\sqrt{n}+j\big)^2-\big(\sqrt{n}+j\big)}{n\big(\sqrt{n}+j\big)}\\
&=\; 1-\frac{2j\sqrt{n}+j^2+\sqrt{n}+j}{n\big(\sqrt{n}+j\big)}\\
&\approx\; 1-\frac{2j\sqrt{n}}{n\sqrt{n}} = 1-2j/n.
\end{aligned}$$

Thus $(1-r_j)/j \approx 2/n$ and so we obtain the following approximation (the factor of 2 is due to the presence of two sums)

$$2\sqrt{\pi n/4}\,t'_{n,0} \;=\; \frac{\sqrt{\pi}\,n!}{n^{\sqrt{n}}(n-\sqrt{n}\,)!},$$

where $(n-\sqrt{n}\,)!$ should be approximated using Stirling's formula (Theorem 1.5 (p. 12)) since we have no formula for $x!$ when x is not a positive integer.

11.4.7. Use Principle 11.6 (p. 349) with $r=1$, $b=0$ and $c=-1$ to obtain

$$a_n \sim n!\,\exp\!\left(-\sum_{k\in S} 1/k\right).$$

11.4.9 (a) The function has radius of convergence $r = 1$ and has a singularity at $-1 = -r$. Another reason we can't use it for $A_e(x)$ is that $a_{e,n} = 0$ whenever n is odd.

(b) For both cases, $r = 1$, $b = 0$ and $c = -1/2$. We obtain $L = \sqrt{2}$ for $A_o(x)$ and $L = 1/\sqrt{2}$ for $A_e(x)$.

(c) By power series, $a_{e,2n} = (-1)^n \binom{-1/2}{n} (2n)!$, which can be rearranged to give the answer. By Stirling's formula, $a_{e,2n} \sim (2n)!/\sqrt{\pi n} \sim 2(2n/e)^{2n}$.

(d) Since $A_o(x) = (1+x)A_e(x)$, we have $a_{o,2n} = a_{e,2n}$ and $a_{o,2n+1} = (2n+1)a_{e,2n}$. By the previous part, $a_{o,2n} = \binom{2n}{n}(2n)!\,4^{-n}$ and $a_{o,2n+1} = \binom{2n}{n}(2n+1)!\,4^{-n}$.

11.4.11. We use Principle 11.6 (p. 349) with

$$A(x) \;=\; (1 - 2x - 3x^2)^{-1/2}, \quad b = 0 \;\text{ and }\; c = -1/2.$$

Since $1 - 2x - 3x^2 = (1 - 3x)(1 + x)$, $r = 1/3$ and

$$L \;=\; \lim_{x \to 1/3} \frac{(1 - 2x - 3x^2)^{-1/2}}{(1 - 3x)^{-1/2}} \;=\; \lim_{x \to 1/3} \frac{1}{\sqrt{1 + x}} \;=\; \frac{\sqrt{3}}{2}.$$

Thus

$$a_n \;\sim\; \frac{\sqrt{3}\; 3^n n^{-1/2}}{2\Gamma(1/2)} \;=\; \frac{3^{n+1/2}}{2\sqrt{\pi n}}.$$

11.4.13. We use Principle 11.6 (p. 349). Since $(1 + x^2)^2 - 4x$ vanishes at $x = r = 0.295597742\ldots$ and is positive whenever $-r \le x < r$, we have found r. Thus we take

$$f(x) \;=\; (1 - x/r)^{1/2}, \qquad g(x) \;=\; \frac{-1}{2}\sqrt{\frac{(1 + x^2)^2 - 4x}{1 - x/r}} \qquad \text{and} \qquad h(x) \;=\; \frac{1 + x^2}{2}.$$

We have

$$L \;=\; \lim_{x \to r} \frac{-\sqrt{(1 + x^2)^2 - 4x}}{2\sqrt{1 - x/r}} \;=\; -\frac{1}{2}\sqrt{\lim_{x \to r} \frac{(1 - x^2)^2 - 4x}{1 - x/r}}.$$

By using l'Hôpital's Rule, we obtain

$$L \;=\; -\frac{1}{2}\sqrt{\frac{-4r(1 - r^2) - 4}{-1/r}} \;=\; -\sqrt{r + r^2(1 - r^2)} \;=\; -0.61265067.$$

11.4.15. By techniques we have used before,

$$H(x) \;=\; x \sum_{k \ge 2} H(x)^k + x.$$

Sum the geometric series and use algebra to obtain the desired quadratic equation for $H(x)$.

This quadratic could be treated as an implicit equation for $H(x)$ and we could apply Principle 11.7 (p. 353). Alternatively, we could solve the quadratic for $H(x)$ and use Principle 11.6 (p. 349). For Principle 11.7, let $F(x, y) = y^2 - y + \frac{x}{1+x}$. Then $F_y(x, y) = 2y - 1$ and so $s = 1/2$ and $\frac{r}{1+r} = 1/2 - (1/2)^2$, which yields $r = 1/3$. For Principle 11.6,

$$H(x) \;=\; \frac{1 - \sqrt{1 - 4x/(1 + x)}}{2} \;=\; \frac{1 - \sqrt{(1 - 3x)/(1 + x)}}{2}.$$

Thus $r = 1/3$, $f(x) = (1 - x/r)^{1/2}$, $g(x) = -1/2(1 + x)^{1/2}$ and $h(x) = 1/2$. In any case, the answer is

$$a_n \sim \frac{\sqrt{3}\, 3^n}{4\sqrt{\pi n^3}}.$$

11.4.17. We will use Principle 11.6 (p. 349). We want to solve

$$\sum_{k\in D} r^k/k! = 1$$

for r because then we have a singularity due to division by zero. This can always be done:

- The radius of convergence of the sum is ∞.
- The sum vanishes at $r = 0$.
- The sum is increasing and unbounded as $r \to +\infty$.

Having found r, we let $f(x) = (1 - x/r)^{-1}$. Then $g(x) = (1 - x/r)A(x)$ and, by l'Hôpital's Rule,

$$L = \lim_{x\to r} \frac{1 - x/r}{1 - \sum_{k\in D} x^k/k!} = \frac{1}{\sum_{k\in D} r^k/(k-1)!}$$

Let $d = \gcd(D)$. You should be able to see that $a_n = 0$ when n is not a multiple of d. Hence we'll need to assume that $d = 1$. (Actually you can get around this by setting $x^d = z$, a new variable.) The answer for general d is

$$a_n \sim \frac{d\, n!\, r^{-n}}{\sum_{k\in D} r^k/(k-1)!} \quad \text{when } d \text{ divides } n$$

and $a_n = 0$, otherwise.

11.4.19 (a) We have

$$B(x) = \sum_{t=0}^{k} \binom{k}{t} f(x)^t g(x)^t h(x)^{k-t}.$$

Apply the principle to each term in the sum. Since $c < 0$, the largest contribution comes from the $t = k$ term. Thus $b_n \sim g(r)^k n^{-ck-1}/\Gamma(-ck)$.

(b) Proceed as in the previous part. Since $c > 0$, the largest contribution is now from $t = 1$ and so $b_n\ ka_n h(r)^{k-1}$.

(c) The formula for $B(x)$ is just a bit of algebra. The only singularity on $[-r, r]$ is due to $f(x) = (1 - x/r)^{1/2}$.

(d) Since $A(x)$ is a sum of nonnegative terms, it is an increasing function of x and so $A(x) = 1$ has at most one positive solution. We take $b = 0$, $c = -1$ and $f(x) = (1 - x/s)^{-1}$ in Principle 11.6. Then

$$\lim_{x\to s} \frac{(1 - A(x))^{-1}}{(1 - x/s)^{-1}} = \lim_{x\to s} \frac{1 - x/s}{1 - A(x)} = \frac{1}{sA'(s)}$$

by l'Hôpital's Rule.

Suppose that $c < 0$. Note that $A(0) = 0$ and that $A(x)$ is unbounded as $x \to r$ because $A(x)/(1 - x/r)^c$ approaches a nonzero limit. Thus $A(x) = 1$ has a solution in $(0, r)$ by the Mean Value Theorem for continuous functions.

(e) If we could deal with $e^{A(x)}$ using Principle 11.6, we could multiply $g(x)$ and $h(x)$ in the principle by $e^{s(x)}$, where $s(x)$ is either of the sums given in this exercise. Then we can apply Principle 11.6.

11.4.21 (b) Let $U(x) = T(x)/x$. The equation for U is $U = \sum x^d U^d/d!$. Replacing x^k with z, we see that this leads to a power series for U in powers of $z = x^k$. Thus the coefficients of x^m in $U(x)$ will be 0 when m is not a multiple of k.

(c) We apply Principle 11.7 (p. 353) with

$$F(x,y) \;=\; y - x\sum_{d\in D} y^d/d! \quad\text{and}\quad F_y(x,y) = 1 - x\sum_{\substack{d\in D\\ d\neq 0}} y^{d-1}/(d-1)!.$$

Using $F(r,s) = 0 = F_y(r,s)$ and some algebra, we obtain

$$\sum_{\substack{d\in D\\ d\neq 0}} (d-1)s^d/d! \;=\; 1 \quad\text{and}\quad r \;=\; \left(\sum_{\substack{d\in D\\ d\neq 0}} s^{d-1}/(d-1)!\right)^{-1}.$$

Once the first of these has been solved numerically for s, the rest of the calculations are straightforward.

Appendix A

A.1. $\mathcal{A}(n)$ is the formula $1+3+\cdots+(2n-1) = n^2$ and $n_0 = n_1 = 1$. $\mathcal{A}(1)$ is just $1 = 1^2$. To prove $\mathcal{A}(n+1)$ in the inductive step, use $\mathcal{A}(n)$:

$$(1+3+\cdots+(2n-1)) + (2n+1) = n^2 + (2n+1) = (n+1)^2.$$

A.3. Let $\mathcal{A}(n)$ be $\sum_{k=1}^{n-1}(-1)^k k^2 \;=\; (-1)^{n-1}\sum_{k=1}^{n} k$. By (A.1), we can replace the right hand side of $\mathcal{A}(n)$ by $(-1)^{n-1}n(n+1)/2$, which we will do. It is easy to verify $\mathcal{A}(1)$. As usual, the induction step uses $\mathcal{A}(n-1)$ to evaluate $\sum_{k=1}^{n-1}(-1)^{k-1}k^2$ and some algebra to prove $\mathcal{A}(n)$ from this.

What would have happened if we hadn't thought to use (A.1)? The proof would have gotten more complicated. To prove $\mathcal{A}(n)$ we would have needed to prove that

$$(-1)^{(n-1)-1}\sum_{k=1}^{n-1} k + (-1)^{n-1}n^2 \;=\; (-1)^{n-1}\sum_{k=1}^{n} k.$$

At this point, we would have to prove this result separately by induction or prove in using (A.1).

A.5. The claim is true for $n = 1$. For $n+1$, we have

$$(x^{n+1})' \;=\; (x^n x)' \;=\; (x^n)'x + (x^n)x' = (nx^{n-1})x + x^n,$$

where the last used the induction hypothesis. Since the right side is $(n+1)x^n$, we are done.

A.7. The inductive step only holds for $n \geq 3$ because the claim that P_{n-1} belongs to both groups requires $n-1 \geq 2$; however, $\mathcal{A}(2)$ was never proved. (Indeed, if $\mathcal{A}(2)$ is true, then $\mathcal{A}(n)$ is true for all n.)

A.9. This is obviously true for $n = 1$. Suppose we have a numbering when $n-1$ lines have been drawn. The nth line divides the plane into two parts, say A and B. Assign all regions in A the same number they had with $n-1$ lines and reverse the numbering in B.

Section B.1

B.1.1. We'll omit most cases where the functions must be nonnegative. Also, the proofs for O properties are omitted because they are like those for Θ without the "A" part of the inequalities. All inequalities are understood to hold for some A's and B's and all sufficiently large n.

(a) Note that $g(n)$ is $\Theta(f(n))$ if and only if there are positive constants B and C such that $|g(n)| \leq B|f(n)|$ and $|f(n)| \leq C|g(n)|$. Let $A = 1/C$. Conversely, if $A|f(n)| \leq |g(n)| \leq B|f(n)|$, let $C = 1/A$ and reverse the previous steps.

(b) These follow easily from the definition.

(c) These follow easily from the definition.

(d) We do Θ using A. Let $A' = A|C/D|$ and $B' = B|C/D|$. Then $A'|Df(n)| \leq |Cg(n)| \leq B'|Df(n)|$.

(e) Use (a): We have $(1/B)|g(n)| \leq |f(n)| \leq (1/A)|g(n)|$.

(f) See proof in the text.

(g) We do $\Theta(\,)$ and use (a). We have $A_i|f_i(n)| \leq |g_i(n)| \leq B_i|f_i(n)|$. Multiplying and using (a) with $A = A_1A_2$ and $B = B_1B_2$ gives the first result. To get the second part, divide the $i = 1$ inequalities by the $i = 2$ inequalities (remember to reverse direction!) and use (a) with $A = A_1/B_2$ and $B = B_1/A_2$.

(h) See proof in the text.

(i) This follows immediately from (h) if we drop the subscripts.

B.1.3. This is not true. For example, n is $O(n^2)$, but n^2 is not $O(n)$.

B.1.5 (a) Hint: There is an explicit formula for the sum of the squares of integers.

(b) Hint: There is an explicit formula for the sum of the cubes of integers.

(c) Hint: If you know calculus, upper and lower Riemann sum approximations to the integral of $f(x) = x^{1/2}$ can be used here.

B.1.7 (a) Here's a chart of values.

	5	10	30	100	300
n^2	25	10^2	9×10^2	10^4	9×10^4
$100n$	5×10^2	10^3	3×10^3	10^4	3×10^4
$100(2^{n/10} - 1)$	41	10^2	7×10^2	10^5	10^8
fastest	A	A, C	C	A, B	B
slowest	B	B	B	C	C

(b) When n is very large, B is fastest and C is slowest. This is because, (i) of two polynomials the one with the lower degree is eventually faster and (ii) an exponential function grows faster than any polynomial.

B.1.9. Let $p(n) = \sum_{i=0}^{k} b_i n^i$ with $b_k > 0$.

(a) Let $s = \sum_{i=0}^{k-1} |b_i|$ and assume that $n \geq 2s/b_k$. We have

$$|p(n) - b_k n^k| \leq \left| \sum_{i=0}^{k-1} b_i n^i \right| \leq \sum_{i=0}^{k-1} |b_i| n^i \leq \sum_{i=0}^{k-1} |b_i| n^{k-1} = s n^{k-1} \leq b_k n^k/2.$$

Thus $|p(n)| \geq b_k n^k - b_k n^k/2 \geq (b_k/2)n^k$ and also $|p(n)| \leq b_k n^k + b_k n^k/2 \leq (3b_k/2)n^k$.

(b) This follows form (a) of the theorem.

(c) By applying l'Hospital's Rule k times, we see that the limit of $p(n)/a^n$ is $\lim_{n\to\infty}(k!/(loga)^k)/a^n$, which is 0.

(d) By the proof of the first part, $p(n) \le (3b_k/2)n^k$ for all sufficiently large n. Thus we can take $C \ge 3b_k/2$.

(e) For $p(n)$ to be $\Theta(a^{Cn^k})$, we must have positive constants A and B such that $A \le a^{p(n)}/a^{Cn^k} \le B$. Taking logarithms gives us $\log_a A \le p(n) - Cn^k \le \log_a B$. The center of this expression is a polynomial which is not constant unless $p(n) = Cn^k + D$ for some constant D, the case which is ruled out. Thus $p(n) - Cn^k$ is a nonconstant polynomial and so is unbounded.

B.1.11 (a) The worst time possibility would be to run through the entire loop because the "`If`" always fails. In this case the running time is $\Theta(n)$. This actually happens for the permutation $a_i = i$ for all i.

(b) Let N_k be the number of permutations which have $a_{i-1} < a_i$ for $2 \le i \le k$ and $a_k > a_{k+1}$. (There is an ambiguity about what to do for the permutation $a_i = i$ for all i, but it contributes a negligible amount to the average running time.) The "`If`" statement is executed k times for such permutations. Thus the average number of times the "`If`" is executed is $\sum kN_k/n!$. If the a_i's were chosen independently one at a time from all the integers so that no adjacent ones are equal, the chances that all the k inequalities $a_1 < a_2 < \cdots < a_k > a_{k+1}$ hold would be $(1/2)^k$. This would give $N_k/n! = (1/2)^k$ and then $\sum_{k=0}^{\infty} kN_k/n!$ would converge by the "ratio test." This says that the average running time is bounded for all n. Unfortunately the a_i's cannot be chosen as described to produce a permutation of $\underline{n}$.

We need to determine N_k. With each arbitrary permutation $a_1, a_2, \ldots$ we can associate a set of permutations $b_1, b_2, \ldots$ counted by N_k. We'll call this the set for $a_1, a_2, \ldots$. For $i > k+1$, $b_i = a_i$, and $b_1, \ldots, b_{k+1}$ is a rearrangement of $a_1, \ldots, a_{k+1}$ to give a permutation counted by N_k. How many such rearrangements are there? b_{k+1} can be any but the largest of the a_i's and the remaining b_i's must be the remaining a_i's arranged in increasing order. Thus there are k possibilities and so the set for $a_1, a_2, \ldots$ has k elements. Hence the set associated with $a_1, a_2, \ldots$ contains k permutations counted by N_k. Since there are $n!$ permutations, we have a total of $n!k$ things counted by N_k; however, each permutation $b_1, b_2, \ldots$ counted by N_k appears in many sets. In fact it appears $(k+1)!$ since any rearrangement of the first $k+1$ b_i's gives a permutation that has $b_1, b_2, \ldots$ in its set. Thus the number of things in all the sets is $N_k(k+1)!$. Consequently, $N_k = n!k/(k+1)!$.

By the previous paragraphs, the average number of times the "`If`" is executed is $\sum k^2/(k+1)!$, which approaches some constant. Thus the average running time is $\Theta(1)$.

(c) The minimum running time occurs when $a_n > a_{n+1}$ and this time is $\Theta(n)$. By previous results the maximum running time is also $\Theta(n)$. Thus the average running time is $\Theta(n)$.

Section B.3

B.3.1 (a) If we have know $\chi(G)$, then we can determine if c colors are enough by checking if $c \ge \chi(G)$.

(b) We know that $0 \le \chi(G) \le n$ for a graph with n vertices. Ask if c colors suffice for $c = 0, 1, 2, \ldots$. The least c for which the answer is "yes" is $\chi(G)$. Thus the worst case time for finding $\chi(G)$ is at most n times the worst case time for the NP-complete problem. Hence one time is O of a polynomial in n if and only if the other is.

Subject Index

The style of a page number indicates the nature of the text material:

the style *123* indicates a definition,
the style 123 indicates an extended discussion, and
the style 123 indicates a short discussion or brief mention.

A

accepting state, *190*.
Adjacent Comparisons Theorem, 241.
admissible flow, *172*.
alcohol, *334*.
Algebra, Fundamental Theorem of, *387*.
algorithm,
  analysis, 367–373.
  greedy, *79*, 151.
  Kruskal's, 151.
  parallel, 239.
  pipelining, *240*.
  Prim's, 151.
  recursive, *207*, *195*.
  running time, 367–373.
  tree traversal, 248–252.
and (logical), *59*.
articulation point, *see* vertex, cut.
asymptotic analysis, *307*.
asymptotic to, *340*.
asymptotically equal, *373*.
Augmentable Path Theorem, 173.
automaton, *see* finite automaton.

B

backtracking, 84–88.
balance, *171*.
balls in boxes, 37, 40, 295.
base case, *198*.
bicomponent, *154*.
Big oh ($O(\ )$ and $O^+(\ )$), *368*.
bijection, *43*.
binary relation, *see* relation, binary.
binary tree, *259*.
binomial coefficient, 19–20, 77.
Binomial Theorem, 20, 273.
Bonferroni's Inequality, 102.
breadth-first,
  edge sequence, *249*.
  order, *249*.
  traversal, *249*.
  vertex sequence, *249*.
buckminsterfullerene, *165*.
Burnside's Lemma, 112–115, 330.

C

canonical, *51*.
capacity, *171*, *172*, *177*.
card hands, 21–23, 68.
Cartesian product, *7*.
Catalan numbers, 15–17, 30, 39, 251, 279–280.
characteristic function, *101*, *115*.
Chebyshev's inequality, 288, 325, 385.
chi ($\chi(S)$), *101*, *115*.
child, *139*, *247*.
chromatic number, 378, *379*.
chromatic polynomial, 158–161.
Church's Thesis, *188*.
clique, *183*.
clique number, *183*.
Codes, error correcting, 27–30.
codomain, *42*.
coimage, *54*.
comparator, *238*.
complement, *200*.
composition of an integer, *10*.
composition, *45*.

conjunctive form, *203*.
connected component, *133*.
convolution, 272.
cut partition, *177*.
cycle, *133*.
  directed, *142*.
  Hamiltonian, *136*.
  index, *333*.
  length, *147*.
  of a permutation, *see* permutation, cycle.
  permutation, *46*.

D

data structure,
  graph representation, 146.
  RP-tree, 264.
  tree traversal, 249.
decision, *66*.
degree of the face, *164*.
DeMorgan's law, *61*.
depth-first,
  edge sequence, *248*.
  postorder traversal, *249*.
  preorder traversal, *249*.
  traversal, 84, *248*.
  vertex sequence, *248*.
derangement, *50*, 99, 283, 318.
derivation, *255*.
descendant, *153*.
digraph, *142*.
  enumerative use, 310–312.
  finite automaton and, 189–192.
  functional, *142*, *329*, *356*.
  simple, *142*.
  strongly connected, *144*.
direct insertion order, 71, 78–79.
direct product, *see* set, direct product.
disjunctive form, *200*.
disjunctive normal form, *62*.
distinct representative, 178–179.
distribution, uniform, *381*.
divide and conquer, *9*, 220, 232.
Dobinski's formula, 320.
domain, *42*.
domino arrangement, 89, 310–312.

E

edge, *66*, *122*, *123*.
  contraction, *159*.
  cut, *134*.
  deletion, *159*.
  parallel, *122*, *125*.
  subgraph, induced by, *133*.
EGF, *316*.
elementary event, *381*.
envelope game, 42.
equations, number of solutions to, 98.
equivalence relation, *see* relation, equivalence.
Error correcting code, 27–30.
error, relative, *12*.
Euler characteristic, *162*.
Euler phi function, *100*, 105.
Euler's relation, 163–164.
Eulerian trail, *145*.
event, *381*.
event, elementary, *381*.
exclusive or, *60*.
expectation of a random variable, *384*.
exponential formula, 321–326.

F

falling factorial, *11*.
father, *139*.
Ferris wheel, 112.
Fibonacci numbers, 34, 36, 224, 275–276, 277.
finite automaton, 189–192.
  deterministic, *191*.
  enumerative use, 310–312.
  grammar of, 191, 256.
  non-d=d, 191.
  nondeterministic, *191*.
  regular sequences and, 304.
finite state machine, *see* finite automaton.
flow, *171*.
flows in network, 170–179.
forest, *326*.
Four Color Theorem, *158*.
full binary tree, *259*.
function, *42*.
  $\lfloor x \rfloor$ (floor), *26*.
  $\lceil x \rceil$ (ceiling), *344*.

function (*continued*):
bijective, *43*.
Boolean, *59*, *200*.
characteristic, *101*, *115*.
codomain of, *42*.
coimage of, *54*.
domain of, *42*.
Euler phi (φ), *100*, 105, *338*.
$\exp(z) = e^z$, *321*.
Gamma (Γ), *349*.
generating, *see* generating function.
graph, incidence, *123*.
image of, *54*.
incidence, *123*.
injective, *43*.
inverse, *43*, *54*.
lex order listing, 76.
monotone, *52*.
monotonic, 51–52.
$\binom{n}{k}$ (binomial coefficient), *19*.
nondecreasing, *51*.
one-line notation for, *42*.
one-to-one, *see* function, injective.
onto, *see* function, surjective.
ordinary generating, *269*.
rank, *see* rank.
recursive, *188*.
restricted growth, *58*.
restriction of, *132*.
strictly decreasing, lex order, 77–78.
surjective, *43*, 97.
two line notation for, *43*.
unrank, *see* rank.
Fundamental Theorem of Algebra, *387*.

G

generating function, 19, 36, *269*.
exponential, *316*.
ordinary, *269*.
grammar,
automaton and, 191.
context-free, *254*.
nonterminal symbol, *254*.
phrase structure, *256*.
production, *254*.
regular, *256*.
regular and finite automaton, 256.
grammar (*continued*):
terminal symbol, *254*.
graph, *123*.
bicomponents of, 154–155.
biconnected, *154*, 167.
bipartite, *134*.
circuit, *135*.
clique number, *183*.
coloring, 157–161, 372, 378.
complete, *160*.
connected, *133*.
directed, *142*.
dual, *162*.
embedding, *149*, *165*.
Eulerian, *135*.
Hamiltonian, *136*.
isomorphism, *128*.
matrix representation of, 146.
multigraph, *124*.
oriented simple, *144*.
planar, *149*, 162–169.
planar, coloring, 165–166.
regular, *164*.
rooted, *138*.
simple, *122*.
st-labeling, *168*.
unlabeled, *125*, 129.
Gray code, 81–82, 86, 220.
group, *112*.
alternating, *49*.
cyclic, *116*, *333*.
dihedral, *116*.
permutation, *112*.
symmetric, *49*.

H

hands of cards, 21–23, 68.
Hanoi, Tower of, 214–216.
heap, *236*.
Heawood's Theorem, 165.
height, *230*.

I

identifier, *255*.
image, *54*.
Inclusion and Exclusion Principle, 94–104.
independent random variables, *383*.
induction, 198–203, 361–365.
induction assumption, *198*, *361*.

induction hypothesis, *198*, *361*.
inductive step, *198*, *361*.
initial condition, *see* recursion, initial condition.
injection, *43*.
input symbol, *190*.
involution, *47*.
isomer, *334*.
isomorphism, graph, *128*.
isthmus, *see* edge, cut.

K

Kuratowski's theorem, 162.

L

l'Hôpital's Rule, *352*.
Lagrange inversion, 326.
language, *254*, *256*.
Language Recognition Problem, *378*.
Latin rectangle, *179*.
Latin square, *88*.
leaf, *139*, *0*, *247*.
lex order, *7*.
  bucket sort and, 234.
  functions listed in, 76.
  listing in, 66.
  permutations listed in, 70.
  strictly decreasing functions listed in, 77.
lexicographic order, *see* lex order.
lineal pair, *153*.
linear ordering, *11*.
  see also list.
list, *5*, *6*.
  basic counting formulae, 40.
  circular, 13, 112.
  unordered, 51–52.
  with repetition, 6–8.
  without repetition, 11–12.
listing,
  by backtracking, 84–87.
  symmetry invariant, *109*.
Little oh ($o(\ )$), *373*.
local description, *213*.
loop, *142*.

M

machine, finite state, *see* finite automaton.
map, *158*.
map coloring, 165–166.
map, coloring, 158.
Marriage Problem, Stable, *194*.
Marriage Theorem, 178.
matrix,
  graph represented by, 146.
  nilpotent, *147*.
  permutation, 48.
Max-Flow Min-Cut Theorem, 177.
maximum flow, *172*.
mean of a random variable, *384*.
Menger's theorem, *180*.
mergesort, *see* sort, merge.
Monte Carlo simulation, *264*.
multinomial coefficient, 23–24, 31.
Multinomial Theorem, 31.
multiplication, polynomial, 343.
multiset, *5*, 36–37, 51–52.

N

necklace, 113.
network flow, 170–179.
node, *see* vertex.
nonterminal symbol, *254*.
notation,
  one-line, *42*.
  postorder, *253*.
  reverse Polish, *253*.
  two-line, *43*.
NP-complete, *see* problem, NP-complete.
$\underline{n}$, *41*.
number,
  Catalan, 15–17, 30, 39, 251, 279–280.
  chromatic, 378, *379*.
  Fibonacci, 34, 36, 199, 204, 224, 275–276, 277.
  partition of, *37*, 295.
  Stirling, *34*, *324*.

O

OGF, *269*.
Oh, big oh ($O(\ )$ and $O^+(\ )$), *368*.
Oh, little oh ($o(\ )$), *373*.
one-line notation, *42*.
or (logical), *59*.
or (logical), exclusive, *60*.
orbit, *330*.
order, linear, *see* list.
output symbol, *190*.

P

palindrome, *9*.
parallel edges, *122*.
parent, *139*.
parsing, *255*.
partial fraction, 276, 387.
partial order, *103*.
partition of a number, *37*, 295.
partition of a set, *see* set partition.
partition, cut, 176–179.
Pascal's triangle, *32*.
path, *66*, *131*.
  augmentable, *173*.
  directed, *142*.
  increment of the, *173*.
  length, *147*.
permutation, *43*, 45–48.
  alternating, *327*.
  cycle form, *46*.
  cycle length, *46*.
  derangement, *50*.
  direct insertion order, 71, 78–79.
  involution, *47*.
  lex order listing, 70.
  parity of, *49*.
  transposition order, 72.
Philip Hall Theorem, 178.
Pigeonhole Principle, 55–57, 58.
pipelining, *240*.
Pólya's Theorem, 333–338.
polynomial multiplication, 343.
polynomial, chromatic, 158–161.
popping, *215*.
postorder,
  edge sequence., *249*.
  notation, *253*.
  traversal, *249*.
  vertex sequence;, *249*.
preorder,
  edge sequence;, *249*.
  traversal, *249*.
  vertex sequence;, *249*.
principal subtree, *247*.
probability space, *381*.
problem,
  assignment, *178*.
  bin packing, *379*.
  coloring, 378.
  halting, *188*.
  intractable, *378*.
  n queens, 90.
  NP-complete, 377–379.
  NP-easy, *378*.
  NP-hard, *378*.
  scheduling, *158*.
  tractable, *378*.
production, *254*, *256*.
programming,
Prüfer sequence, *141*.
pushing, *215*.

Q

queens problem, 90.
queue, *249*.

R

Ramsey Theory, 201.
random variable, *382*.
  expectation, *384*.
  independence, *383*.
  mean, *384*.
  variance, *384*.
rank, *70*.
  all functions in lex order, 77.
  formula for, 76.
  permutations in direct insertion order, 79.
  strictly decreasing functions in lex order, 78, 87.
  subsets of given size, 78, 87.
recursion, 32–35, *197*, 224.
  constant coefficient, 282–283.
  initial condition, *32*.
  linear, *341*.
recursive,
  approach, *204*, *195*.
  definition, *204*.
  formula, *204*.

recursive (*continued*):
solution, *204*.
regular sequence, *304*.
counting, 296.
relation,
binary, *103*, *127*, *142*.
covering, *145*.
equivalence, 103, *111*, 126–130, 154.
residual tree of, *75*.
reverse Polish notation, *253*.
root, *138*, *247*.
rooted map, *330*.
RP-tree, *139*, *247*.
and parsing, 255.
binary, 248–266, 288–289.
binary, counting, 279–280.
breadth-first traversal, *249*.
depth-first traversal, *248*.
number of leaf, 286–288.
postorder traversal, *249*.
preorder traversal, *249*.
restricted, counting, 353–354.
statistic, 264–265.
unlabeled, 259–266.
Rule of Product, *6*, 9–10, 292–298, 317–320.
Rule of Sum, *8*, 9–10, 292–298, 317–320.

S

sample, *6*.
SDR (system of distinct representatives), 178.
selection, *6*.
sentence, *254*.
sequence, *6*.
sequence, counting, 296, 304, 318.
series,
see also function, generating.
alternating, 99.
bisection of, *274*.
geometric, *271*, 273.
multisection of, *274*.
set, *5*, 19–23, 51–52.
cut, 176–179.
direct product, *41*.
incomparable, 25.
partially ordered, *103*.
set partition, *34*, 54.
block, *34*.
Dobinski's formula, 320.
partition, block, 54.
sibling, *139*.
sink, *172*.
son, *139*, *247*.
sort,
see also sorting network.
binary insertion, *233*.
bucket, *234*.
comparisons, lower bound on, 229.
divide and conquer approach, 221.
Heapsort, *236*.
insertion, *233*.
merge, 206, 210–212, *235*.
merge, time, 278.
methods, 227–228.
Quicksort, *235*.
Quicksort, time, 289–290, 350.
sorting network, *238*.
Adjacent Comparisons Theorem, 241.
Batcher, 235, 239, 243–244, 308.
brick wall, *239*, 244.
Bubble, *239*.
Zero-One Principle, 241.
source, *172*.
space, probability, *381*.
Sperner family, *25*.
Stable Marriage Problem, *194*.
stack, *15*, *215*, *249*.
standard deviation, *384*.
start symbol, *254*.
starting state, *189*.
state transition table, *189*.
Stirling number,
Stirling numbers of the first kind, *50*, *324*.
Stirling numbers of the second kind, *34*, 54, 97.
Stirling's formula ($n!$), *12*, 39.
string, *6*.
string, counting, 296, 304, 318.
subgraph, *132*.
directed, *142*.
induced by vertex, *133*.
subset, 77, 82, 86, 87.
sum, rapidly decreasing terms, *345*.
sum, slowly decreasing terms, *345*.
surjection, *43*, 97.

syntax, *254*.

T

Taylor's Theorem, *267*, 270.
terminal symbol, *254*.
theorems (some),
  Adjacent Comparisons Theorem, 241.
  Augmentable Path Theorem, 173.
  Binomial Theorem, 20.
  Bonferroni's inequality, 102.
  Burnside's Lemma, 112.
  Chebyshev's Inequality, 385.
  convolution formula, 272.
  Dobinski's formula, 320.
  exponential formula, 321.
  Five Color (Heawood's) Theorem, 165.
  Four Color Theorem, 158.
  Fundamental Theorem of Algebra, 387.
  Kuratowski's Theorem, 162.
  Lagrange inversion, 326.
  Marriage (Philip Hall) Theorem, 178.
  Max-Flow Min-Cut Theorem, 177.
  Multinomial Theorem, 31.
  Pigeonhole Principle, 55.
  Pólya's Theorem, 333.
  Principle of Inclusion and Exclusion, 95.
  Ramsey's Theorem, 202.
  Rule of Product, 6.
  Rule of Sum, 8.
  Stirling's formula ($n!$), 12.
  Taylor's Theorem, 270.
  Vandermonde's formula, 274.
  Zero-One Principle, 241.
Theta ($\Theta(\ )$), *368*.
Tower of Hanoi, 214–216.
trail, *131*.
transposition order, 72.
Traveling Salesman Problem, *378*.
traversal, 248–252.
tree, *66*, *136*, *139*.
  binary, *140*, *259*.
  decision, *65*.
  depth first spanning, *153*.
  free, *136*.
  full binary, *140*, *259*.
tree (*continued*):
  heap, *236*.
  height of rooted, *140*.
  height of, *230*.
  labeled, counting, 143–144.
  leaves of, *66*.
  lineal spanning, *153*.
  ordered rooted, *66*.
  parse, *255*.
  postorder traversal, *249*.
  preorder traversal, *249*.
  root of, *66*.
  rooted labeled, counting, 326.
  rooted plane, *see* RP-tree.
  rooted unlabeled, counting, 354–355.
  spanning, *139*, 150–154.
  spanning, minimum weight, 150–154.
  traversal algorithms, 84, 248–252.
truth table, *61*.
Turing machine, *188*.
two-line notation, *43*.

U

uniform distribution, *381*.
unrank, *see* rank.

V

value, *172*.
Vandermonde's formula, *274*.
variable, random, *382*.
variance of a random variable, *384*.
Venn diagram, *96*.
vertex, *66*, *122*, *123*.
  adjacent, *123*.
  cut, *134*.
  degree, *124*, *126*.
  isolated, *125*, *132*.
  loop, *125*.
  sequence, *131*.
  sink, *172*.
  source, *172*.
  terminal (leaf), *140*.
VLSI design, 169.

W

walk, *131*, *147*.
weakly decreasing, *52*.
word, *6*.

Z

Zero-One Principle, 241.